WIND RIVER

# VxWorks®

# DEVICE DRIVER DEVELOPER'S GUIDE
## Volume 3: Legacy Drivers and Migration

6.6

**Corporate Headquarters**
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

**http://www.windriver.com**

For information on how to contact Customer Support, please visit the following URL:

**http://www.windriver.com/support**



6 Nov 07
Part #: DOC-16146-ND-00

# Contents


1 Introduction ..... 1

1.1 Legacy Driver Overview ..... 1

1.2 Before You Begin ..... 2

1.3 About This Documentation ..... 2

Navigating this Documentation Set ..... 2

2 Adding an Existing Driver to Your BSP ..... 3

2.1 Introduction ..... 3

2.2 BSP Support for Legacy (Non-VxBus) Device Drivers ..... 4

2.3 Project Facility ..... 4

2.4 Component Descriptor Files ..... 5

3 END Ethernet Drivers ..... 7

3.1 Introduction ..... 7

3.2 END Driver Overview ..... 8

3.2.1 Driver Environment ..... 8

The MUX ..... 8
Network Interface Drivers and Protocols ..... 9
The MUX, Protocol, and Driver API ..... 10
Driver Components ..... 12
Protocols That Use the MUX API ..... 13
Interactions With the MUX API ..... 17
Network Layer to Data Link Layer Address Resolution ..... 23

3.2.2 VxWorks OS Interface ..... 24

Understanding How VxWorks Launches and Uses Your Driver ..... 24
Executing Calls Waiting In the Network Job Queue ..... 28
Adding Your Network Interface Driver to VxWorks ..... 29
Allocating, Initializing, and Utilizing Memory Resources ..... 31
Handling Packet Reception ..... 39
Handling Packet Transmission ..... 52
Implementing Checksum Offloading ..... 58
Implementing Required Entry Points and Structures ..... 58

**3.3 The END Driver Development Process ..... 81**

3.3.1 Driver Development Overview ..... 81

Writing a New Driver ..... 81
Porting an Existing Driver From Another OS ..... 83
Additional Development Issues ..... 83

3.3.2 Error Conditions ..... 84

3.3.3 Generic MIB Interface Initialization ..... 86

**4 SCSI Drivers ..... 93**

**4.1 Introduction ..... 93**

**4.2 SCSI Overview ..... 94**

4.2.1 Layout of SCSI Modules ..... 95

4.2.2 The VxWorks OS Interface ..... 98

Libraries ..... 98
Driver Programming Interface ..... 101

**4.3 SCSI BSP Interface ..... 132**

**4.4 The SCSI Driver Development Process ..... 135**

**4.5 Common SCSI Driver Development Issues ..... 135**

4.5.1 Troubleshooting and Debugging ..... 135

4.5.2 Test Suites ..... 136

scsiDiskThruputTest( ) ..... 137
scsiDiskTest( ) ..... 137
scsiSpeedTest( ) ..... 139
tapeFsTest( ) ..... 139

**5 Timestamp Drivers ..... 141**

**5.1 Introduction ..... 141**

**5.2 Timestamp Driver Overview ..... 142**

5.2.1 Hardware Environment ..... 142

5.2.2 VxWorks OS Interface ..... 146

Working with the Wind River System Viewer ..... 147
Timestamp Driver Components ..... 148
Sample Drivers ..... 148

**5.3 Timestamp Driver Configuration and BSP Interface ..... 163**

sysTimestampConnect( ) ..... 163
sysTimestampEnable( ) ..... 164
sysTimestampDisable( ) ..... 164
sysTimestampPeriod( ) ..... 164
sysTimestampFreq( ) ..... 165
sysTimestamp( ) ..... 165
sysTimestampLock( ) ..... 165

**5.4 The Timestamp Driver Development Process ..... 166**

5.4.1 Timers that Can Be Read While Enabled ..... 166

Timer Period ..... 166
Interrupt Level ..... 167
Interrupt Locking ..... 167

5.4.2 Working Around Deficiencies In Hardware Timers ..... 167
Timer Re-Synchronization ..... 167
Timer Period ..... 168
Down Counter ..... 168
Counter Preloading ..... 168
Adjustment for Time Skew ..... 168
Counter Read Optimization ..... 169
5.4.3 Using the VxWorks System Clock Timer ..... 169
Timer Rollover Interrupt ..... 169
Timer Counter Not Reset ..... 169
Timer Period ..... 170
**5.5 Common Timestamp Driver Development Issues ..... 170**
**6 Additional Drivers ..... 171**
**6.1 Introduction ..... 171**
**6.2 ATAPI Drivers ..... 172**
**6.3 Interrupt Controller Drivers ..... 172**
BSP Interface ..... 172
Non-Vectored Interrupt Sources ..... 173
**6.4 Memory Drivers ..... 174**
6.4.1 Hardware Mismatches ..... 174
6.4.2 Complex Modern Memory Controllers ..... 174
**6.5 Multi-Mode (SIO) Serial Drivers ..... 176**
6.5.1 SIO_CHAN and SIO_DRV_FUNCS ..... 176
6.5.2 Polled Mode, WDB, and Kernel Initialization ..... 179
6.5.3 Serial Ports, WDB, and Interrupts ..... 179
**7 Migrating to VxBus ..... 181**
**7.1 Overview ..... 181**

**7.2 Porting an Existing VxWorks Driver to VxBus ..... 181**

7.2.1 Verifying Your Hardware and Driver Code ..... 182

7.2.2 Creating the VxBus Infrastructure ..... 182

Driver Source File ..... 183
Driver Header Files (Optional) ..... 183
Driver Component Description File ..... 183
Driver Configuration Stub Files ..... 184
Modifying the BSP (Optional) ..... 185
Verifying the infrastructure ..... 186

7.2.3 Moving Existing Code into the New Source File ..... 187

7.2.4 Removing Driver Code from the BSP ..... 188

7.2.5 Adding Debug Code ..... 188

7.2.6 Changing Initialization to VxBus ..... 189

7.2.7 Adding VxBus Driver Methods ..... 190

7.2.8 Updating Names within the Source File ..... 190

7.2.9 Removing BSP Dependencies ..... 191

7.2.10 Converting Register Access in Existing Code ..... 192

7.2.11 Removing Global Variables ..... 193

**Index ..... 195**

# 1 Introduction


1.1 Legacy Driver Overview 1

1.2 Before You Begin 2

1.3 About This Documentation 2


## 1.1 Legacy Driver Overview

The term *legacy driver* is used to describe pre-VxBus device drivers as implemented in early VxWorks 6.*x* and in VxWorks 5.*x* releases. Unlike VxBus model device drivers, legacy drivers do not share a common interface to the operating system or hardware.

Legacy drivers continue to be supported in this release (for uniprocessor systems only). However, many drivers and BSPs distributed for this release have been updated to take advantage of the VxBus infrastructure. (For information on VxBus, see *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*).

**NOTE:** Legacy device driver implementations are valid for uniprocessor (UP) systems only. If you intend to use VxWorks in symmetric multiprocessor (SMP) mode, you must implement VxBus model device drivers for your system. (For information on SMP, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*).

## 1.2 Before You Begin

Wind River strongly recommends that you develop new VxWorks device drivers according to the VxBus model whenever possible. Before beginning your device driver development, consider which device driver model you will implement. Be sure to read and understand the information provided in this chapter and *7. Migrating to VxBus*. Also be sure to read and understand the information provided in the early chapters of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*. This information can help you make an educated decision about which driver model you need to implement for your development. It can also help you to successfully navigate and understand this documentation set.

## 1.3 About This Documentation

The information in this document does not apply to new development. The legacy driver information provided in this chapter is for the purpose of maintaining existing legacy device driver code. The driver-specific chapters of this document may not provide sufficient information for developing new drivers according to the legacy device driver model. In particular, the networking information provided in these sections may be insufficient for new driver development. If you require more information on the Wind River Network Stack, see the Wind River Network Stack documentation provided with this release.

In addition to the legacy information provided for maintenance purposes, this volume provides information on migrating an existing legacy model driver to the VxBus model. Wind River recommends that you migrate your legacy driver code to the VxBus device driver model when possible. For more information, see *7. Migrating to VxBus*.

#### Navigating this Documentation Set

For information on navigating this documentation set, documentation conventions, and other available documentation resources, see *VxWorks Device Driver Developer's Guide (Vol. 1): Getting Started with Device Driver Development*.

# 2

# Adding an Existing Driver to Your BSP


2.1 Introduction 3

2.2 BSP Support for Legacy (Non-VxBus) Device Drivers 4

2.3 Project Facility 4

2.4 Component Descriptor Files 5


## 2.1 Introduction

A driver is considered properly integrated into the VxWorks code base when it can be included in a system configuration either by defining a macro in the BSP **config.h** file or by including a component in a project. Integration requires more than just placing the driver in the appropriate directory. It also entails:

- Providing BSP support.
- Integrating the driver with the appropriate configuration facilities (GUI and command line).
- Providing appropriate component description file (CDF) entries. (For information on CDF files, see the *VxWorks Kernel Programmer's Guide: Kernel*.)

## 2.2 BSP Support for Legacy (Non-VxBus) Device Drivers

Drivers are included in BSPs in several ways. For typical legacy drivers, it is expected that the driver itself resides in a file in *installDir*/**vxworks-6.***x*/**target/src/drv/***type*. For example, the driver for 16550 serial ports is in *installDir*/**vxworks-6.***x*/**target/src/drv/sio/ns16550Sio.c**.

Each BSP must provide an access layer that allows the driver to be used regardless of the location of the device registers. This code is kept in a **sys***Dev***.c** file in the BSP directory. For example, to use the **ns16550Sio.c** file, the BSP contains a file named **sysNs16550Sio.c**. In some cases, the entire driver is contained in the **sys***Dev***.c** file.

**NOTE:** You can also include the translation layer directly in **sysLib.c**, although this is not the preferred method. In general, you should keep all device-specific code out of **sysLib.c**.

For some devices, certain parts of the initialization code must be put into **sysLib.c**. The BSP routine **sysHwInit( )** is responsible for setting all devices to a *quiescent* state. That is, the device does not generate interrupts when interrupts are enabled. For many devices, the power-on behavior is such that the device is initialized to a quiescent state. If this is not the case, **sysHwInit( )** needs to either quiesce the device itself or call a routine (contained in **sys***Dev***.c**) to quiesce the device.

Typically, the device-driver file is included in **sysLib.c** by file inclusion based on preprocessor macros. For example, in the **wrPpmc7400** BSP, support for the 16550 serial device is contained in the **sysSerial.c** file. The **sysSerial.c** file is included from **sysLib.c**, and the **ns16550Sio.c** file is included in **sysSerial.c**. This two-step method allows support for serial devices to be separated from generic board code in **sysLib.c**, but also allows the driver object code to be included in **sysLib.o**.

## 2.3 Project Facility

For VxWorks 6.*x*, information about integrating device drivers into the project facility can be found in the *VxWorks Kernel Programmer's Guide*.

## 2.4 Component Descriptor Files



For the driver to be selectable in the Wind River development suite environment (Workbench), there must be an entry for it in a CDF file and this entry must be brought into the project facility folder hierarchy. CDF files reside in *installDir*/**vxworks-6.***x*/**target/config/comps/vxWorks** and are parsed by the project facility in alphabetical order. The files are written in component description language (CDL) which is described in the *VxWorks Kernel Programmer's Guide: Kernel*.

# 3 END Ethernet Drivers


3.1 Introduction 7

3.2 END Driver Overview 8

3.3 The END Driver Development Process 81


## 3.1 Introduction

**NOTE:** The information in the chapter is provided for reference purposes only. You should use this information to maintain existing END Ethernet driver code. If you want to develop a new network driver, see *VxWorks Device Driver Developers Guide, Volume 1* and *VxWorks Device Driver Developer's Guide (Vol. 2): Network Drivers*.

A network interface driver written especially for use with the network stack is known as an enhanced network driver (or END driver). This chapter describes how to write an END driver. It also provides information on how END drivers interact with VxWorks and certain networking protocols.

This chapter assumes that you are a software developer familiar with general networking principles, including protocol layering. Familiarity with 4.4 BSD networking internals is also helpful. This chapter is not a tutorial on writing network interface drivers. Instead, you should use this chapter as a guide for writing a network interface driver that runs under VxWorks.

**NOTE:** The *installDir*/**vxworks-6.***x*/**target/src/drv/end** directory contains a **templateEnd.c** file.

## 3.2 END Driver Overview

This section discusses how an END driver interfaces with VxWorks and how it differs from other network drivers. The section also includes a discussion of the components that make up an END driver.

**NOTE:** The networking information provided in this chapter is also legacy information. For current information on the Wind River Network Stack, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide* volumes.

### 3.2.1 Driver Environment

This section discusses the various elements of the END driver environment, including the MUX and MUX layers, and END driver components.

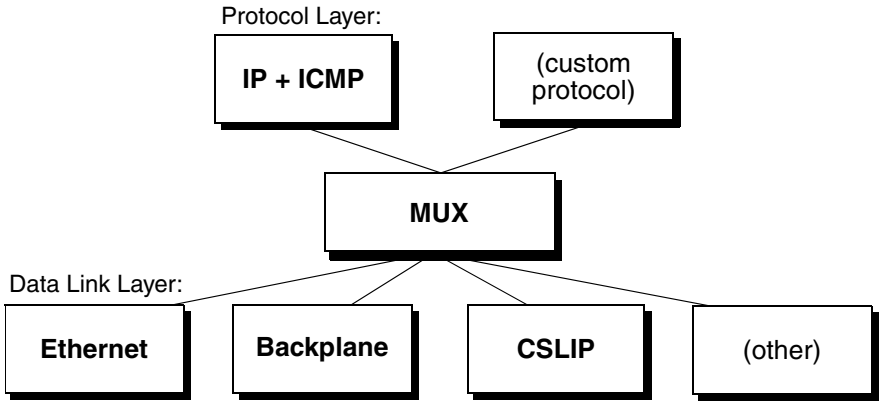
#### The MUX

The multiplexor (usually known as the *MUX*, and referred to as the MUX in this document) is an interface that joins the data link and protocol layers. An END driver does not directly interface with the data link layer, but rather interfaces with the MUX, which is an abstraction layer that is intended to de-couple the END driver from any particular protocol. This API multiplexes access to the networking hardware for multiple network protocols. Figure 3-1 shows the MUX in relationship to the protocol and data link layers.

At the protocol layer, VxWorks typically uses IP, although other network protocols can be ported to VxWorks. At the data link layer, VxWorks typically uses Ethernet, although it does support other physical media for data transmission. For example, VxWorks supports the use of serial lines for long-distance connections. In more closely coupled environments, VxWorks internet protocols can also use the shared memory on a common backplane as the physical medium. However, whatever the

medium, the network interface drivers all use the MUX to communicate with the protocol layer.

Figure 3-1 **The MUX Interface Between Data Link and Protocol Layers**




**NOTE:** The data link layer is an abstraction. A network interface driver is code that implements the functionality described by that abstraction. Likewise, the protocol layer is an abstraction. The code that implements the functionality of the protocol layer could be called a protocol interface driver. However, this document refers to such code simply as "the protocol."

### Network Interface Drivers and Protocols

Using the BSD 4.3 model, VxWorks network drivers and protocols are tightly coupled. Both the protocol and the network driver depend on an intimate knowledge of each other's data structures. Under the MUX-based model, network drivers and protocols have no knowledge of each other's internals. Network interface drivers and protocols interact only indirectly, through the MUX.

For example, after receiving a packet, the network interface driver does not directly access any structure within the protocol. Instead, when the driver is ready to pass data up to the protocol, the driver calls a MUX-supplied routine. This routine then handles the details of passing the data up to the protocol.

The purpose of the MUX is to de-couple the network driver from the network protocols, thus making the network driver and network protocols nearly independent from each other. This independence makes it much easier to add new

drivers or protocols. For example, if you add a new END driver, all existing MUX-based protocols can use the new driver. Likewise, if you add a new MUX-based protocol, any existing END driver can use the MUX to access the new protocol.

### The MUX, Protocol, and Driver API

Figure 3-1 shows a protocol, the MUX, and a network interface driver. The protocol implements the following entry points:

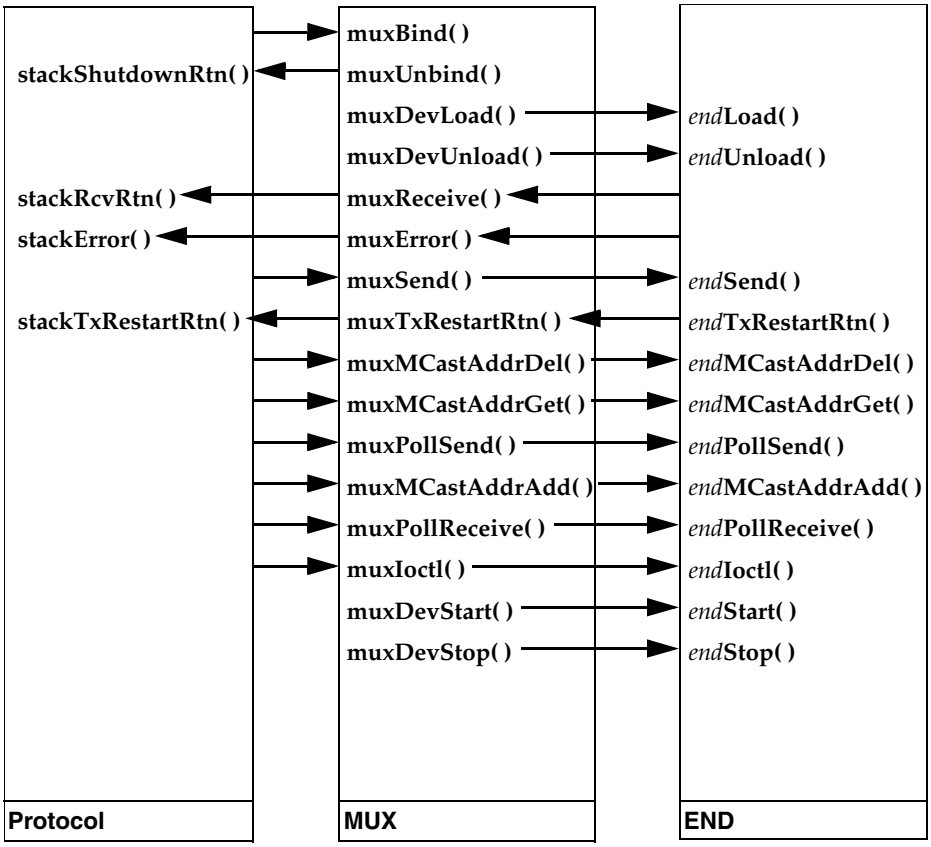
- **stackShutdownRtn( )**
- **stackError( )**
- **stackRcvRtn( )**
- **stackTxRestartRtn( )**

The MUX calls these entry points when it needs to interact with a protocol. To port a protocol to use the MUX, you must implement some or all of the entry points listed above (some protocols may omit certain entry points).

The MUX implements the entry points **muxBind( )**, **muxUnbind( )**, **muxDevLoad( )**, and so forth. Both the protocol and the driver call the MUX entry points as needed. Because the MUX is already implemented, it requires no additional coding work from the developer.

The network interface driver implements the entry points *end***Load( )**, *end***Unload( )**, *end***Send( )**, and so forth. The MUX uses these entry points to interact with the network interface driver. When writing or porting a network interface driver to use the MUX, you must implement all of the entry points listed in Table 3-2 in *Required Driver Entry Points*, p.65.

Figure 3-2 **The MUX Interface**




In Figure 3-2, the arrows indicate calls to an entry point. For example, the top-most arrow tells you that the protocol calls **muxBind( )**, a routine implemented in the MUX. If the MUX-based API specifies both ends of the call, the figure specifies a routine name at each end of an arrow. For example, **muxSend( )** calls *end***Send( )**. Note that although the protocol started the send by calling **muxSend( )**, the figure does not name the protocol routine that called **muxSend( )**. That routine is outside the standardized API.

### Driver Components

**NOTE:** The prevalent model of network interface devices available today is the direct memory access (DMA) engine. This document assumes the use of devices that are DMA engines. If you are developing a driver for a device that uses programmed I/O or some other proprietary shared memory technique, the DMA-specific portions of this text may not be directly applicable to your driver.

An END driver's basic components include:

- a receiver
- a transmitter
- a command and control module

The receiver is composed of the routines that execute an algorithm to:

- accept incoming frames from a DMA (direct memory access) engine
- pass the incoming frames to the MUX
- provide the DMA engine with a continuous supply of DMA buffers

The transmitter is composed of the routines that execute an algorithm to:

- accept packets from the MUX and transfer them to the device's transmit DMA engine
- reclaim the resources associated with a transmitted packet

The command and control module provides configuration, initialization, and control interfaces for the device.

An END driver receiver is stimulated by a device-generated interrupt. The driver does not directly service incoming frames in the interrupt's context but defers the work to a routine run in a task context.

Each instance of an END driver has a private buffer pool into which incoming DMAs are directed. An END driver loans individual buffers from its pool to the stack. There is no guarantee that the network stack returns the loaned buffers to the END driver.

The larger the END driver operating bandwidth, the greater its memory requirements. Occasionally, an END driver does not have sufficient memory resources to accommodate the data inflow. This can be due to system constraints, buffer loaning, or CPU starvation. When a driver gets into an insufficient resource condition, it continues to provide the DMA engine buffers into which inflowing data is transferred but the driver does not pass these buffers up to the stack.

A protocol requests that an END driver transmit a frame by calling the **muxSend( )** routine, which in turn calls the driver's registered send routine. Sends can occur at any time, and may occur before previous sends have completed.

Resource reclamation of DMA buffers and control structures is generally stimulated by a device-generated transmit-packet-complete interrupt. This interrupt announces that the device has sent a complete frame and that the driver can now return the memory resources back to the pool. In many cases, this interrupt occurs excessively. Therefore, in order to improve performance, you must reduce the frequency of packet-complete interrupts. However, take care to ensure that you reliably return memory resources to the pool. If a device does not provide a packet-complete interrupt, then the driver must use its own means to ensure resource reclamation.

A stall condition occurs when the device determines that it has momentarily exhausted its resources. The stall can occur in either the receiver or the transmitter. When a stall occurs, the device halts operations in the module in which it detected the stall. To resume operation, sufficient resources must be reclaimed and made available. Often a device register must also be cleared.

The END driver command and control module is the part of the driver that parses the driver configuration parameters, quiesces the device, and configures the device in the prescribed mode. It incorporates the driver's load, unload, start, stop, and **ioctl( )** routines, as well as routines for querying and modifying the multicast filter. In essence, the driver's command and control provides the driver's external interface, with the exception of send and receive. This includes the driver interrupt service routine, which should be considered a part of the driver command and control module.

Interrupts alert the driver to packets received, packet transmit DMA completion, and stall, error, or link state change conditions.

### Protocols That Use the MUX API

This section describes how to port protocols to the MUX-based model. As shown in Figure 3-1, MUX-based protocols bind themselves to the MUX from above and network interface drivers (END drivers) bind themselves to the MUX from below. Thus, a protocol is layered on top of the MUX, which is layered on top of a network interface driver. The responsibilities of each are summarized below.

Protocol:

- Interface to the transport layer, and through it, to the application programs.
- Usually, acts as a source of transmit packets and a sink of received packets.
- Returns buffer resources from received packets to the driver pools.

MUX:

- Calls driver load, unload, start, stop, and other control routines.
- Binds and unbinds protocols.
- Delivers packets received by an END driver to the appropriate bound protocols.
- Calls protocol transmit restart routines when requested by the END driver.

Network interface driver:

- Deals with hardware.
- Loads (allocates and initializes) the driver's END interface objects and buffer pools.
- Unloads (terminates and frees) the driver's END interface objects and buffer pools.
- Delivers received packets to the MUX.
- Transmits packets and frees associated buffer resources.

A protocol writer has to deal only with calls to the MUX. Everything device-specific is handled in the drivers of the data link layer—the layer below the MUX.

### Protocol Startup

Each protocol that wants to receive packets must first attach to a network interface. To do this, the protocol calls **muxBind( )**. The returned routine value is a cookie that identifies the END device to which the MUX has bound the protocol. The protocol must save this cookie for use in subsequent calls to the MUX.

As input to **muxBind( )**, you must specify the base name and unit number of a network device (for example, **ln** and **0**, **ln** and **1**, **ei** and **0**, and so on), as well as the appropriate receive, transmit restart, and shutdown routines for the protocol; a protocol type, and a name for the attaching protocol.

There are three special protocol type values, as well as the normal network-layer protocol type values from RFC 1700, corresponding to the Ethernet header **type** field. The three special type values are **MUX_PROTO_OUTPUT**, **MUX_PROTO_SNARF**, and **MUX_PROTO_PROMISC**. **MUX_PROTO_OUTPUT** is used for output protocols—which are passed packets in the send path, but not the receive path. There may be no more than one output protocol for a given interface. (Output protocols are discussed further below). **MUX_PROTO_SNARF** protocols, normal "typed" protocols, and **MUX_PROTO_PROMISC** protocols attached to an END interface may be delivered packets received on that interface.

When the END driver passes a received packet to the MUX, it includes a pointer to the **END_OBJ** structure representing the interface. This structure contains pointers to an array of (non-output) protocols bound to the interface. Snarf protocols, those with type **MUX_PROTO_SNARF**, are placed first in the array and are passed every received packet that is not consumed by an earlier snarf protocol. (The WDB agent using the **WDB_COMM_END** communication strategy, and the Berkeley Packet Filter (BPF), are examples of snarf protocols.) After the snarf protocols, the array lists normal "typed" protocols such as IPv4 (0x0800), ARP (0x0806), and IPv6 (0x86dd). There may be only one such protocol of a given type bound to a given interface. The MUX delivers a packet to one of these protocols only if it is not consumed by a snarf protocol, and the packet's type matches the protocol's type. Promiscuous protocols, those that specify the type **MUX_PROTO_PROMISC**, occur last in the array and are delivered any packets not consumed by a snarf protocol, a normal typed protocol, or an earlier promiscuous protocol.

A protocol consumes a packet by returning **TRUE** (or any non-zero value) from its receive routine; it is responsible for freeing the packet. A protocol that does not consume a packet passed to its receive routine should not modify or free the packet.

**NOTE:** The presence of snarf protocols can decrease the receive performance for all typed protocols. Also, among normal typed protocols, those whose packets are most common on the network (or most performance-critical in a particular system) should be bound first (if possible) to ensure the best performance.

#### Output Protocols

A single protocol can be bound to each device for the filtering of output packets. This functionality is provided for applications that want to look at every packet that is output on a particular device. The type **MUX_PROTO_OUTPUT** is passed into **muxBind( )** when this protocol is registered. Only the **stackRcvRtn( )** parameter is valid with this type.

### Sending Data

To put the appropriate address header information into the buffer, the protocol calls **muxAddressForm( )**. Finally, to send the packet, the protocol calls **muxSend( )**, passing in the cookie returned from the **muxBind( )** as well as the **mBlk** that contains the packet it wants to send. The MUX then hands the packet to the driver.

### Receiving Data

In response to an interrupt from the network device, VxWorks executes the device's previously registered interrupt service routine. This routine gets the packet off the device and queues it for processing the task level, where the driver prepares the packet for hand-off to the MUX. For a more detailed description of this process, see *Handling Packet Reception*, p.39.

To hand the packet off to the MUX, the driver calls **muxReceive( )**. The **muxReceive( )** routine determines the protocol type of the packet (0x800 for IP, 0x806 for ARP, and so on) and then searches its protocol list to see if any have registered using this protocol type.

If there is a protocol that can handle this packet, the MUX passes the packet into the **stackRcvRtn( )** specified in the protocol's **muxBind( )** call. Before passing the packet to a numbered protocol (that is, a protocol that is neither a **MUX_PROTO_SNARF** nor a **MUX_PROTO_PROMISC** protocol) **muxReceive( )** calls the **muxPacketDataGet( )** routine and passes two **mBlk**s into the protocol.

The first **mBlk** contains all the link-level information. The second **mBlk** contains all of the information that comes just after the link-level header. This partitioning of the data lets the protocol skip over the header information (it also breaks the BSD 4.3 model at the **do_protocol_with_type( )** interface). The protocol then takes over processing the packet.

This new method of multiplexing received packets eliminates the method based on the **etherInputHook( )** and **etherOutputHook( )** routines. If a protocol wants to see all of the undeliverable packets received on an interface, it specifies its type as **MUX_PROTO_PROMISC**.

If a protocol needs to modify data received from the network, it should copy that data first. Because other protocols might also want to see the raw data, the data should not be modified in place (that is, in the received buffer).

### Protocol Transmission Restart

The **muxTkSend( )** routine may return an error, **END_ERR_BLOCK**, indicating that the network driver has insufficient resources to transmit data. The network service sublayer can use this feedback to establish a flow control mechanism by holding off on making any further calls to **muxTkSend( )** until the device is ready to restart transmission. At that time, the MUX calls the *stack***RestartRtn( )** that you registered for the interface at bind time.

**NOTE:** Such a flow control mechanism must be implemented in the network service sublayer. It is not provided by the MUX implementation.

### Protocol Shutdown

When a protocol is finished using an interface, or for some reason wants to shut itself down, it calls the **muxUnbind( )** routine. This routine tells the MUX to deallocate the **NET_PROTOCOL** and other memory allocated specifically for the protocol.

## Interactions With the MUX API

This section presents the routines and data structures that the protocol uses to interact with the MUX. Most of the work is handled by the MUX routines (listed in Table 3-1). Unlike the driver entry points described earlier, you do not implement the MUX routines. These routines are utilities that you can call from within your protocol. For specific information on these MUX routines, see the appropriate API reference entry.

These MUX routines do not comprise the entire MUX/protocol interface. In addition, a protocol must implement a set of standardized routines that handle things such as shutting down the protocol, restarting the protocol, passing data up to the protocol, and passing error messages up to the protocol.

Table 3-1 **MUX Interface Routines**

| MUX Routine | Purpose |
|---|---|
| **muxDevLoad( )** | Loads a device into the MUX. |
| **muxDevStart( )** | Starts a device from the MUX. |
| **muxBind( )** | Hooks a protocol to the MUX. |

Table 3-1 **MUX Interface Routines** (cont'd)

| MUX Routine | Purpose |
|---|---|
| **muxSend( )** | Accepts a packet from the protocol and passes it to the device. |
| **muxDataPacketGet( )** | Gets an **mBlk** containing packet data only. The link-level header information is omitted. |
| **muxAddressForm( )** | Forms an address into an outgoing packet. |
| **muxIoctl( )** | Accesses control routines. |
| **muxMCastAddrAdd( )** | Adds a multicast address to the list maintained for a device. |
| **muxMCastAddrDel( )** | Deletes a multicast address from the list maintained for a device. |
| **muxMCastAddrGet( )** | Gets the multicast address table maintained for a device. |
| **muxUnbind( )** | Disconnects a protocol from the MUX. |
| **muxDevStop( )** | Stops a device. |
| **muxDevUnload( )** | Unloads a device. |
| **muxPacketDataGet( )** | Extracts the packet data (omitting the link-level data) from a submitted **mBlk** and writes it to a fresh **mBlk**. |
| **muxPacketAddrGet( )** | Extracts source and destination address data (omitting the packet data) from a submitted **mBlk** and writes each address to its own **mBlk**. If the local source/destination addresses differ from the end source/destination addresses, this routine writes to as many as four **mBlk**s. |
| **muxTxRestart( )** | If a device unblocks transmission after having blocked it, this routine calls the **stackTxRestartRtn( )** routine associated with each interested protocol. |
| **muxReceive( )** | Sends a packet up to the MUX from the device. |
| **muxShutdown( )** | Shuts down all protocols above this device. |

Table 3-1 **MUX Interface Routines** (cont'd)

| MUX Routine | Purpose |
|---|---|
| **muxAddrResFuncAdd( )** | Adds an address resolution function to the address resolution function list. |
| **muxAddrResFuncGet( )** | Gets a particular address resolution function from the list. |
| **muxAddrResFuncDel( )** | Deletes a particular address resolution function from the list. |



### The Protocol Data Structure NET_PROTOCOL

For each protocol that binds to a device, the MUX allocates a **NET_PROTOCOL** structure. The MUX uses this structure to store information relevant to the protocol, such as the protocol's type, its receive routine, and its shutdown routine. These are chained in a linked list whose head rests in the **protocols** member of the **END_OBJ** structure the MUX uses to manage a device. The **NET_PROTOCOL** structure is defined in **end.h** as follows:

```
typedef struct net_protocol
    {
    NODE node;                  /* How we stay in a list. */
    char name[32];              /* String name for this protocol. */
    long type;                  /* Protocol type from RFC 1700 */
    int flags;                  /* Is protocol in a promiscuous mode? */
    BOOL (*stackRcvRtn) (void *, long, M_BLK_ID, M_BLK_ID, void*);
                                /* The routine to call when we get */
                                /* a packet. */
    STATUS (*stackShutdownRtn) (void*, void*);
                                /* The routine to call to shutdown */
                                /* the protocol stack. */
    STATUS (*stackTxRestartRtn) (void*, void*);
                                /* Callback for restarting on blocked tx. */
    void (*stackErrorRtn) (END_OBJ*, END_ERR*, void*);
                                /* Callback for device errors. */
    void* pSpare;               /* Spare pointer that can be passed to */
                                /* the protocol. */
    } NET_PROTOCOL;
```

### Passing a Packet Up to the Protocol: stackRcvRtn( )

Each protocol must provide the MUX with a routine that the MUX can use to pass packets up to the protocol. This routine must take the following form:

```
void stackRcvRtn
    (
    void*         pCookie,    /* returned by muxBind() call */
    long          type,       /* protocol type from RFC 1700 */
    M_BLK_ID      pNetBuff,   /* packet with link level info */
    LL_HDR_INFO*  pLinkHdr,   /* link-level header info structure */
    void*         pSpare      /* a void* the protocol can use to get info */
                              /* on receive. This was passed to muxBind().*/
    )
```

Your protocol must declare its **stackRcvRtn( )** as void. Thus, this routine returns no value.

The parameters are:

**pCookie**
Expects the pointer returned from the **muxBind( )** call. This pointer identifies the device to which the MUX has bound this protocol.

**type**
Expects the protocol type from RFC 1700 or the SAP.

**pNetBuff**
Expects a pointer to an **mBlk** structure that contains the packet data and the link-level information.

**pLinkHdr**
Returns an **LL_HDR_INFO** structure containing header information that is dependent upon the particular data-link layer that the END driver implements. For more information, see *Tracking Link-Level Information: LL_HDR_INFO*, p.63.

**pSpare**
Expects a pointer to the spare information (if any) that was passed down to the MUX using the **pSpare** parameter of the **muxBind( )** call. This information is passed back up to the protocol by each **receiveRtn( )** call. The use of this information is optional and protocol-specific.

### Passing Error Messages Up to the Protocol: stackError( )

The MUX uses the **stackError( )** routine to pass error messages from the device to the protocol. Your code for this routine must have an appropriate response for all possible error messages. The prototype for the **stackError( )** routine is as follows:

```
void stackError
    (
    END_OBJ* pEnd,   /* pointer to END_OBJ */
    END_ERR* pError, /* pointer to END_ERR */
    void* pSpare     /* pointer to protocol private data passed in muxBind */
    )
```

You must declare your **stackShutdownRtn( )** as returning void. Thus, there is no returned function value for this routine. The parameters are:

**pEnd**
Expects the pointer returned as the function value of the **muxBind( )** for this protocol. This pointer identifies the device to which the MUX has bound this protocol.

**pError**
Expects a pointer to an **END_ERR** structure, which **end.h** defines as follows:

```
typedef struct end_err
    {
    INT32 errCode; /* error code, see above */
    char* pMesg;   /* NULL-terminated error message, can be NULL */
    void* pSpare;  /* pointer to user defined data, can be NULL */
    } END_ERR;
```

Within your code for the **stackError( )** routine, you must have appropriate responses to the flags stored in the **errCode** member. Wind River reserves the lower 16 bits of **errCode** for its own error messages, which are as follows:

**END_ERR_INFO**
This error is information only.

**END_ERR_WARN**
A non-fatal error has occurred.

**END_ERR_RESET**
An error occurred that forced the device to reset itself, but the device has recovered.

**END_ERR_DOWN**
A fatal error occurred that forced the device to go down. The device can no longer send or receive packets.

**END_ERR_UP**
The device was down but is now up again and can receive and send packets.

The upper 16 bits of the **errCode** member are available to user applications. Use these bits to encode whatever error messages you need to pass between drivers and protocols.

**pSpare**
Expects a pointer to protocol-specific data. Originally, the protocol passed this data to the MUX when it called **muxBind( )**. This data is optional and protocol-specific.

### Shutting Down a Protocol: stackShutdownRtn( )

The MUX uses **stackShutdownRtn( )** to shut down a protocol. Within this routine, you must do everything necessary to shut down your protocol in an orderly manner. Your **stackShutdownRtn( )** must take the following form:

```
void stackShutdownRtn
    (
    void*  pCookie     /* Returned by muxBind() call. */
    void*  pSpare      /* a void* that can be used by the protocol to get */
                       /* info on receive. This was passed to muxBind().*/
    )
```

You must declare your **stackShutdownRtn( )** as returning void. Thus, there is no returned function value for this routine.

The parameters are:

**pCookie**
Expects the pointer returned as the function value of the **muxBind( )** for this protocol. This pointer identifies the device to which the MUX has bound this protocol.

**pSpare**
Expects the pointer passed into **muxBind( )** as **pSpare**.

### Restarting Protocols: stackTxRestartRtn( )

The MUX uses the **stackTxRestartRtn( )** to restart protocols that had to stop transmitting because the device was out of resources. In high-traffic situations, a **muxSend( )** can return **END_ERR_BLOCK.** This error return indicates that the device is out of resources for transmitting more packets and that the protocol should wait before trying to transmit any more packets.

When the device has determined that it has enough resources to start transmitting again, it can call the **muxTxRestart( )** routine, which, in turn, calls the protocol's **stackTxRestartRtn( )**.

Your **stackTxRestartRtn( )** must take the following form:

```
void muxTxRestart
    (
    void* pCookie /* Returned by muxBind() call. */
    )
```

The parameters are:

**pCookie**
Expects the pointer returned as the function value of the **muxBind( )** for this protocol. This pointer identifies the device to which the MUX has bound this protocol.



#### Network Layer to Data Link Layer Address Resolution

The MUX provides several routines for adding network layer to data link layer address resolution functions. Resolving a network layer address into a data link layer address is usually carried out by a separate protocol. In most IP over Ethernet environments this is carried out by ARP (the address resolution protocol).

Using the MUX, any protocol/data link can register its own address resolution function. The functions are added and deleted by the following pair of routines:

```
STATUS muxAddrResFuncAdd
    (
    long ifType,          /* Media interface type from m2Lib.h */
    long protocol,        /* Protocol type from RFC 1700 */
    FUNCPTR addrResFunc   /* Function to call. */
    )
STATUS muxAddrResFuncDel
    (
    long ifType,          /* Media interface type from m2Lib.h */
    long protocol         /* Protocol type from RFC 1700 */
    )
```

These routines add and delete address resolution routines. The protocol writer is expected to ascertain the exact arguments to that routine. Currently, the only address resolution routine provided by Wind River is **arpresolve( )**.

To find out what address resolution routine to use for a particular network/datalink pair, call the following routine:

```
FUNCPTR muxAddrResFuncGet
    (
    long ifType,          /* ifType from m2Lib.h */
    long protocol         /* protocol from RFC 1700 */
    )
```

This routine returns a pointer to a routine that you can call to resolve data link addresses for the network protocol specified as the second argument.

## 3.2.2 VxWorks OS Interface

This section discusses how END drivers interface with VxWorks including information on how VxWorks launches your driver, how to add your driver to VxWorks, and how to deal with memory resources. It also includes information on sending and receiving packets.

**NOTE:** This section discusses use of the task, **tNetTask**. In VxWorks 6.6, this task is replaced by **tNet0**.

#### Understanding How VxWorks Launches and Uses Your Driver

The primary focus of this section is on the MUX utilities and the standard END driver entry points. However, when designing or debugging your driver's entry points, you need to know the context in which the entry point executes. Thus, you need to know the following:

- The task that makes the calls that actually load and start your driver.
- The task that typically registers the interrupt handler for your driver.
- The task that uses your driver to do most of the processing on a packet.

##### Launching Your Driver

At system startup, VxWorks spawns the task **tUsrRoot** to handle the following:

- Initializing the network task's job queue.
- Spawning **tNetTask** to process items on the network task's job queue.
- Calling **muxDevLoad( )** to load your network driver.
- Calling **muxDevStart( )** to start your driver.

###### Loading Your Driver into the MUX

To load your network driver, **tUsrRoot** calls **muxDevLoad( )**. As input to the call, **tUsrRoot** specifies your driver's *end***Load( )** entry point. Internally, the **muxDevLoad( )** call executes the specified *end***Load( )** entry point.

The *end***Load( )** routine handles any device-specific initialization and returns an **END_OBJ** structure. Your *end***Load( )** routine must populate most of this structure (see *Providing Network Device Abstraction: END_OBJ*, p.58). This includes providing

a pointer to a **NET_FUNCS** structure populated with function pointers to your driver's entry points for handling sends, receives, and so forth.

The *end***Load( )** routine handles parameter parsing, configuration, and initialization. A list of the driver parameters is passed to the *end***Load( )** routine. The routine first allocates memory for the driver control structure and passes a pointer to the driver control structure. It then passes the driver parameters to a parser that breaks the parameters down into discrete values and loads them into the driver control structure.

*end***Load( )** configures the device's registers either as the default configuration or as prescribed by the driver parameters. *end***Load( )** calls a memory initialization routine that allocates a contiguous amount of memory for DMA descriptors, the amount allocated is determined by the number of descriptors specified in the parameters or a default amount defined in the driver. The memory initialization routine also calls **netPoolCreate( )** in **netBufLib** causing it to create a memory pool sufficient for the driver's needs.

The memory initialization routine initializes the driver DMA descriptors. It accesses each discrete descriptor and fills the descriptor fields according to the device expectations and the driver parameter directions. In the case of receive descriptors, it also obtains a tuple from the **netPool** it created, writes the tuple cluster pointer into the descriptor, and stores the tuple **mBlk** pointer in a convenient location from which it can later be correlated back to the descriptor DMA buffer.

**NOTE:** A *tuple* is a construct used by the VxWorks stack and drivers to access and manage data buffers. A detailed description of a tuple is provided in *Receive and Transmit Descriptors*, p.32.

After control returns from *end***Load( )** to **muxDevLoad( )**, the MUX completes the **END_OBJ** structure (by giving it a pointer to a routine your driver can use to pass packets up to the MUX). The MUX then adds the returned **END_OBJ** to a linked list of **END_OBJ** structures. This list maintains the state of all currently active network devices on the system. After control returns from **muxDevLoad( )**, your driver is loaded and ready to use.

#### Registering Your Driver's Interrupt Routine

To register your driver's interrupt handler, you must call **sysIntConnect( )**. The most typical place to make this call is in your driver's **endStart( )** entry point. When **muxDevLoad( )** loads your driver, it calls **muxDevStart( )**, which then calls your driver's **endStart( )** entry point.

### Using tNetTask

When working with END drivers, it is necessary to understand the use of **tNetTask**, how it operates, and why to use it.

Many desktop and mainframe operating systems use network drivers that dispatch incoming packets directly to the application that receives the packets. This operation is done in the lower half of the OS, from within interrupt context. Therefore, much of the network stack is executed from within interrupt service routines (ISRs).

Because VxWorks is intended for real-time applications, ISRs must be kept short. Wind River does not recommend use of long ISRs for network packet processing. For this reason, most of the network stack processing for incoming packets—processing that would typically be done from within an ISR—is pushed to a task context in VxWorks. **tNetTask** is the task that handles this network processing.

#### Interrupt Handlers

Upon arrival of an interrupt on the network device, VxWorks invokes your driver's previously registered interrupt service routine. Your interrupt service routine should do the minimum amount of work necessary to get the packet off the local hardware. To minimize interrupt lock-out time, your interrupt service routine should handle only those tasks that require minimal execution time, such as error or status change. Your ISR should queue all time-consuming work for processing at the task level.

Aside from the general practice of limiting the amount of work done in an ISR, in VxWorks, it is not possible to directly call the MUX receive entry point from an ISR. Instead, it must be called from a task context.

To queue packet-reception work for processing at the task level, your ISR must call **netJobAdd( )**. As input, this routine accepts a function pointer and up to five additional arguments (parameters to the routine referenced by the function pointer).

```
STATUS netJobAdd
    (
    FUNCPTR    routine,    /* routine to add to netTask work queue */
    int        param1,     /* first arg to added routine */
    int        param2,     /* second arg to added routine */
    int        param3,     /* third arg to added routine */
    int        param4,     /* fourth arg to added routine */
    int        param5      /* fifth arg to added routine */
    )
```

In your call to **netJobAdd( )**, you should specify your driver's entry point for processing packets at the task level. The **netJobAdd( )** routine then puts the

function call (and arguments) on the **tNetTask** work queue. VxWorks uses **tNetTask** to handle task-level network processing.

**NOTE:** You can use **netJobAdd( )** to queue up work other than processing for received packets.



**CAUTION:** Use **netJobAdd( )** sparingly. The **netJobRing** is a finite resource that is also used by the network stack. If it overflows, this implies that the network stack is corrupted.

There are several limitations on network interrupts in VxWorks. These limitations impact the way drivers are written.

The interrupt handler generally serves three functions. These functions include:

- handling receive interrupts
- returning resources to the pool after a transmitted packet
- handling error conditions

Network devices typically provide a single interrupt line for all types of interrupts. When an interrupt service routine is called, the ISR must check a register to see what type of action is required. The ISR reads the device register and invokes the appropriate routines to handle each type of exception that has occurred. This invocation is typically accomplished by using calls to **netJobAdd( )** for transmit interrupts, receive interrupts, and to handle error conditions. This means that the interrupt handler itself is short, because most of the work is done in the task-level handlers.

The task-level routines for each type of interrupt should process all the work that is available for that particular type. If all the work of a given type is processed, no subsequent interrupts of that type are required until the service routine is finished. For performance reasons, interrupts for each type of service should be disabled before dispatching a routine with **netJobAdd( )**.

At the time that a driver is started, the physical interface should be activated and the initialized state should be enabled for all interrupts the driver services. Interrupts for specific types of actions should be disabled until the task-level handler has determined that all work of the type associated with that interrupt is complete. When the task-level handler is finished all work for a specific type of interrupt, the interrupt should be re-enabled.

**netJobRing** has a limited amount of space. Because of this, it is critical that the driver make efforts to conserve space on the **netJobRing**. If the ring is allowed to overflow, the network stack can become corrupt and the system may require a

reboot. To safeguard against overflow, the driver must limit the number of jobs that it simultaneously places on the ring. This limit can be imposed through the use of queuing indicators. These indicators communicate to the driver if a particular interrupt handler is already queued on the ring. If the handler is already queued, it is not practical to queue it again before it has run. The indicators are fields in the driver control structure. There should be one indicator for the receive handler and another for the packet-complete interrupt. These indicators are discussed further in *Receive Handler Interlocking Flag*, p.44 and *Transmit-Packet-Complete Handler Interlocking Flag*, p.52, respectively.

- **Interrupt Masking**

For maximum performance, a task-level interrupt handler should be written in such a way as to continue to handle its work until there is no more work outstanding. The ISR should only be executed if the task-level handler is not active. Continuing to execute the ISR while the task-level handler is running hinders performance by interrupting the system for work that is already scheduled. After the first interrupt schedules a task-level handler, the incident interrupt is masked by its ISR and is unmasked just before its task-level handler exits.

### Executing Calls Waiting In the Network Job Queue

The **tNetTask** task sleeps on an incoming work queue. In response to an incoming packet, your ISR calls **netJobAdd( )**. As parameters to **netJobAdd( )**, your interrupt service routine specifies your driver's entry point for handling task-level packet reception. The **netJobAdd( )** call adds this entry point to **tNetTask**'s work queue. The **netJobAdd( )** call also automatically gives the appropriate semaphore for awakening **tNetTask**.

Unless there is a high priority task running, **tNetTask** runs immediately after the ISR completes. Upon awakening, **tNetTask** de-queues function calls and associated arguments from its work queue. It then executes these functions in its context. The **tNetTask** task runs as long as there is work on its queue. When the queue is empty and all packets have been successfully handed off to the MUX, **tNetTask** goes back to sleep on the queue. In this way, processing of incoming packets in VxWorks is handled in the context of **tNetTask**. This prevents network processing from severely interfering with high priority tasks, especially real-time tasks.

It is possible to design a driver that starves the network stack and other drivers. When a driver uses **taskDelay( )**, or any other delay mechanism, in code executed in the context of **tNetTask**, the delay prevents processing of packets from other interfaces. For this reason, you must carefully consider the use of delays in the

driver. Consider rescheduling the job with another **netJobAdd( )** call instead of delaying with **taskDelay( )**. This allows other interfaces, as well as the network stack, to perform other work while the driver is waiting.

Because interrupts are relatively costly in terms of overall system performance, one recommended goal of network device drivers is to process as many packets as possible before exiting. However, to avoid starvation of other interfaces, there should be a cap on the number of packets processed at any one time. If additional packets are available when the cap is reached, the driver can re-schedule the receive routine with another call to **netJobAdd( )**.

### Adding Your Network Interface Driver to VxWorks

Adding your driver to the target VxWorks system is much like adding any other application. The first step is to compile and include the driver code in the VxWorks image. For a description of the general procedures, see the *Wind River Workbench User's Guide*, as well as the *VxWorks Kernel Programmer's Guide*. These documents provide information on how to compile source code to produce target-suitable object code.

In addition to including the object module in the VxWorks image, you must do some additional work to initialize the END driver and get the MUX to recognize it.

All Wind River VxWorks 6.*x* BSPs support an END driver. However, if the BSP you are using does not already include END driver support, you need to create a table of configuration information for END drivers, called **endDevTbl[ ]**. Once this is accomplished, you must populate the table with information about your driver and make sure your BSP calls the appropriate initialization routines. This is usually done in the file **configNet.h** in the BSP directory.

It is also necessary to create definitions containing the configuration information. This is typically done with **#define** statements, grouped together in one location in **configNet.h**. You can get a sample of this table from a reference or template BSP.

Initialization is done from within the routine **usrNetInit( )** in the default system initialization code. By default, **usrNetInit( )** is called based on whether the macros **INCLUDE_NETWORK** and **INCLUDE_NET_INIT** are defined. The BSP needs to have these defined in order for the driver to be included and initialized. These macros are usually defined in **config.h**.

If the BSP already supports an END driver, the BSP should already contain the **endDevTbl[ ]** and appropriate macros. In this case, the **endDevTbl[ ]** table must be modified to include the new driver and you must create definitions containing

the configuration information (this is typically done with **#define** statements, grouped together in one location in **configNet.h**).

In addition, VxWorks drivers are typically written to be independent of the bus and processor configuration. This means that the methods used to access device registers are provided by the BSP and not by the driver. For each supported driver, there is typically a **sys***Dev***.c** file containing the definitions and routines necessary for the driver to get access to the device registers, interrupt connection code, and other resources. When adding a new driver to a BSP, this file must be provided.

For example, if you want VxWorks to create two network devices, one that supports buffer loaning and one that does not, you would first edit **configNet.h** to include the following statements:

```
/* Parameters for loading the driver supporting buffer loaning. */
#define LOAD_FUNC_0 ln7990EndLoad
#define LOAD_STRING_0 "0xffffffe0:0xfffffffe2:0:1:1"
#define BSP_0 NULL

/* Parameters for loading the driver NOT supporting buffer loaning. */
#define LOAD_FUNC_1 LOAD_FUNC_0
#define LOAD_STRING_1 "0xffffee0:0xffffffee2:4:1:1"
#define BSP_1 NULL
```

To set appropriate values for these constants, consider the following:

**END_LOAD_FUNC**
Specifies the name of your driver's *end***Load( )** entry point. For example, if your driver's *end***Load( )** entry point is **fei82557EndLoad( )**, you must edit **config.h** to include the line:

```
#define END_LOAD_FUNC fei82557EndLoad
```

**END_LOAD_STRING**
Specifies the initialization string passed into **muxDevLoad( )** as the **initString** parameter.

**CAUTION:** Each END driver defines the parameters contained in **END_LOAD_STRING** differently. Check the driver carefully to determine what parameters are contained in the load string, and in what order they are expected.

You must also edit the definition of the **endTbl** (a table in **configNet.h** that specifies the END drivers included in the image) to include the following:

```
END_TBL_ENTRY endTbl
    {
    { 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, FALSE},
    { 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, FALSE},
    { 0, END_TBL_END, 0, NULL},
    };
```



The number at the beginning of each line specifies the unit number for the device. The first line specifies a unit number of 0. Thus, the device it loads is *deviceName*0. The **FALSE** at the end of each entry indicates that the entry has not been processed. After the system has successfully loaded a driver, it changes this value to **TRUE** in the run-time version of this table. If you want to prevent the system from automatically loading your driver, set this value to **TRUE**.

Finally, you must edit the BSP **config.h** file to define **INCLUDE_END**.[1] This tells the build process to include the END/MUX interface. A this point, you are ready to rebuild VxWorks to include your new drivers. When you boot this rebuilt image, it calls **muxDevLoad( )** for each device specified in the table in the order listed.

### Allocating, Initializing, and Utilizing Memory Resources

There are five types of memory allocation associated with an END driver. The considerations and requirements differ for each type of memory, depending on several factors. The types of memory allocation include:

- memory allocated for the driver control structure
- memory allocated for receive and transmit descriptors
- memory allocated for the association list
- memory used for **mBlk**s and **clBlk**s
- memory used for cluster buffers

#### Driver Control Structure

Because a device driver must be able to control multiple instances of a device within the same system, it cannot use global variables that pertain to a specific instance of a device. To cope with this limitation, END drivers collect their instance variables into a driver control structure. The driver allocates and initializes a unique structure for each instance of a device under control. Memory allocation for the driver's control structure has no restrictions other than it must be zeroed before any fields are initialized and it should always be cached.

1. By default, the **config.h** file for BSPs that support END drivers undefine **INCLUDE_END**.

### Receive and Transmit Descriptors

The control constructs shared by the device and driver are the descriptors that compose the receive ring and the transmit queue.

The device uses the descriptors to:

- locate DMA buffers
- pass filled buffers to or from the device
- communicate DMA status between the device and the driver software

A descriptor includes a pointer to a DMA buffer. The device DMA engine reads the buffer address from the descriptor and then reads or writes data into or out of the DMA buffer.

A DMA engine always uses a physical address while the software uses a virtual address. It is the driver's responsibility to convert a buffer's virtual address to a physical address. The conversion of a virtual to physical address is, in most cases, a simple process. However, the conversion of a physical address back to a virtual address is more difficult. The driver must store the buffer's original virtual address in a way that can be readily correlated back to the physical address. Therefore, the driver needs to maintain both physical and virtual addresses. This can be especially difficult due to the large number of buffers and their transitory association with descriptors.

The solution to this virtual and physical address storage issue is provided by the tuple. The tuple is a construct that consists of an **mBlk** structure, a **clBlk** structure, and a cluster buffer. The **mBlk** is similar in nature to the **mbuf** used in the BSD network stack. The **mBlk** has a **pClBlk** field, which is a pointer to the **clBlk**. The **clBlk** in turn holds a pointer to the cluster buffer. The cluster buffer is the DMA buffer. The **mBlk** also has a pointer to the cluster buffer but this pointer can be modified by software to add or subtract offsets. The cluster buffer pointer in the **clBlk** always points to the base of the cluster buffer. This provides a convenient place for the driver to store a DMA buffer's virtual address. This scheme depends on the permanence of the tuple constructs. The access path to a cluster buffer in a tuple is **pMblk->pClBlk->clNode.pClBuf**.

Receive and transmit descriptors must not be cached unless there is special snooping provided by the hardware device. If the device requires any alignment restrictions, the descriptors must conform to them.

It is desirable to combine the allocations of receive and transmit descriptors into one allocation. Performance is improved by combining descriptor allocations into one memory block because it reduces the number of TLB misses.

**NOTE:** Because it appears to create more readable source code, driver developers are often tempted to write the driver in such a way that it forces a structure onto the descriptor instead of using offsets. However, if your driver is expected to operate with multiple architectures, accessing the descriptors through a structure is problematic. When accessing a descriptor, the device always accesses the descriptor fields by using offsets from the base address of the descriptor. A driver must access the same exact locations (relative to the descriptor's base address) as the device. Because compilers are allowed to manipulate the size, placement, and even order of different fields in a structure, it is not easy to determine the exact location required. It is not possible to guarantee the behavior of all compilers with regard to structure layout. Therefore, it is impossible to guarantee that a structure will layout exactly the same way across multiple architectures. For this reason, using a structure to access descriptor fields or device registers is not recommended for drivers that are intended to port across architectures. Instead use offsets to access descriptor fields. For more information, see *Wind River Coding Conventions*.

#### Initializing and Utilizing Transmit and Receive Descriptors

The exact organization and properties of a driver's transmit and receive descriptors are determined by the device's specification.

Transmit descriptors are typically organized as a pair of lists—a free list and a transmit queue. All transmit descriptors are initially on the free list. When a descriptor is used to send data through the device, it is transferred to the transmit queue. When its data has been sent, the descriptor is returned to the free list. When a descriptor is first initialized or returned to the free list, it has no associated data and its fields are set to indicate it is available for use. When a descriptor is to be used, it is associated with data to be sent, its fields are set to indicate it has data to be sent, and it is transferred to the transmit queue.

Receive descriptors are typically organized as a ring. Both the device and the driver follow this ring and use or service the ring's descriptors, respectively. The driver follows the device's access, servicing the descriptors the device uses. When the device uses a descriptor, it sets the descriptor’s fields to indicate that its associated buffer has received DMA data. When the driver services the descriptor, it removes the filled buffer and replaces it with an empty one. The driver obtains the replacement buffer from its pool. It then clears the descriptor to indicate to the device that the descriptor is again ready for use. When the ring is first initialized, all descriptors have empty buffers and are ready for use.

**NOTE:** A complication to buffer replacement is that some architectures only read data on a four-byte boundary. An Ethernet header is 14 bytes long. If a DMA buffer is four-byte aligned, then the IP header is two-byte aligned. This results in an alignment mismatch. To compensate for this issue, the driver can offset the DMA buffer pointer in the descriptor by two bytes in order to put the IP header, and subsequent data, at four-byte boundaries. There is a further complication in that this solution requires the device to restrict DMA to a two byte address alignment. Not all devices support DMA using a two byte alignment. Therefore, a device that cannot perform a DMA write to a two-byte boundary cannot work with an architecture that cannot read from a two-byte boundary without copying the data to a new buffer to adjust the packet alignment.

#### Association List

DMA descriptors only store the cluster buffer pointer. Because the buffer has no pointer to either the **clBlk** or the **mBlk**, it is the responsibility of the driver to maintain the correlation between the cluster buffer and its tuple.

The tuple association problem is solved by an *association list*. This technique is enabled by the fact that the receive DMA descriptors are allocated in contiguous memory. This means that no matter how the device accesses the descriptors, either as an array or a linked list, the driver can always access them as an array. The driver keeps an index that increments through the set of descriptors and rolls over between the last and first items. For example:

```
index = (++index % numRxDesc);
```

This allows the driver to use the descriptor index to cross-reference another array that holds the tuples' associated **mBlk** pointers. The driver passes the **mBlk** pointer from the association buffer to the stack. The driver places the **mBlk** pointer from the new tuple into the association list before it increments the index.

The association list should be allocated from cached memory and must be zeroed before initialization.

### Setting Up and Using Memory for Receive and Transmit Buffers

This section describes how **mBlk**, **clBlk**, and cluster buffer elements (collectively known as a tuple) are used in END drivers. The section also provides guidelines for setting up a memory pool.

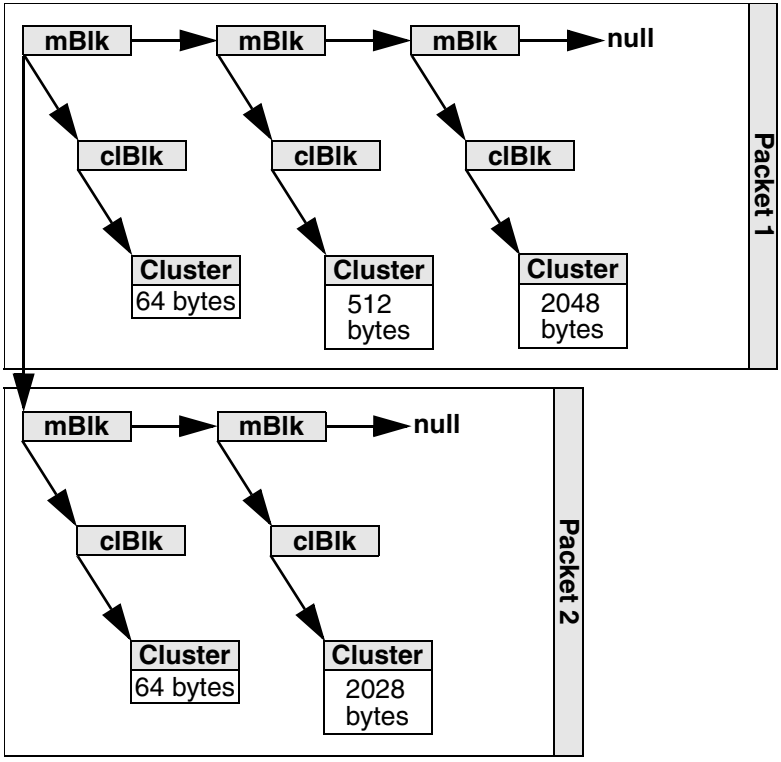
#### mBlks, clBlks, and Cluster Buffers

Included with the network stack is **netBufLib**, a library that you can use to set up and manage a memory pool specialized to the buffering needs of networking

applications such as END drivers and network protocols. To support buffer loaning and other features, **netBufLib** routines deal with data in terms of **mBlk**s, **clBlk**s, and clusters.

The **netBufLib** routines use the **mBlk** and **clBlk** structures to track information necessary to manage the data in the clusters. The clusters contain the data described by the **mBlk** and **clBlk** structures. These elements—**mBlk**s, **clBlk**s, and cluster buffers—constitute a tuple. The **mBlk** structure is the primary vehicle through which you access or pass the data that resides in a tuple. Because an **mBlk** merely references the data, this lets network layers communicate data without actually having to copy the data. Another **mBlk** feature is chaining. This lets you pass an arbitrarily large amount of data by passing the **mBlk** at the head of an **mBlk** chain. See Figure 3-3.

Figure 3-3 **Presentation of Two Packets to the TCP Layer**




The **netBufLib** library provides two means of creating a network memory pool—the routines **netPoolInit( )** and **netPoolCreate( )**. The routines differ in that

**netPoolInit( )** requires the user to allocate the memory used for the tuples. **netPoolCreate( )** takes as arguments, attributes describing the characteristics of the pool's memory and allocates and manages the memory on behalf of the user. This is a great advantage because it provides the driver with properly aligned and cacheable cluster buffers. Wind River highly recommends that you use **netPoolCreate( )** instead of **netPoolInit( )**.

When you use the **netPoolCreate( )** routine to create a net pool, you have the option to use a default set of underlying routines or to use an alternate set of underlying routines. With the default routine set, the **netPoolCreate( )** routine constructs the tuples each time they are needed and de-constructs them each time they are reclaimed. This default behavior is retained for backward compatibility with **netPoolInit( )**. However, Wind River now provides an alternate routine set, **_pLinkPoolFuncTbl**, that implements atomic tuples. That is, that the base tuples are permanently constructed and maintained as an indivisible—or atomic—construct. This reduces unnecessary overhead.

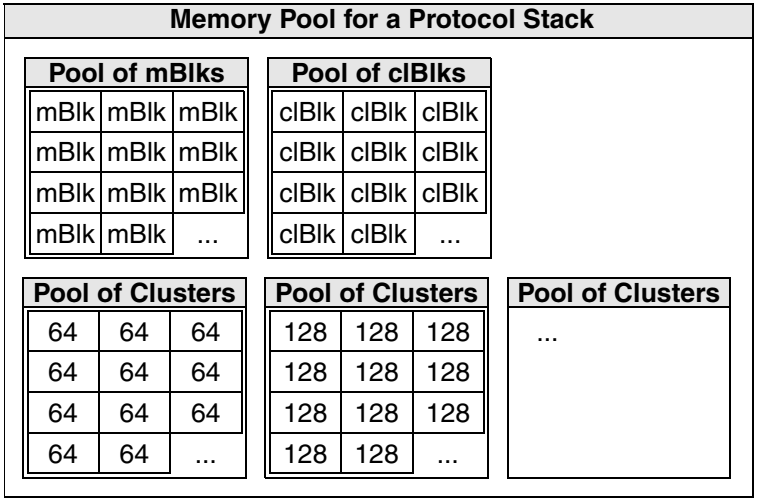
If your device does not allow you to use the provided memory-management utilities, you can write replacements. However, your replacements must conform to the **netBufLib** API for these routines.

#### Setting Up a Memory Pool

Each END driver unit requires its own memory pool. How you configure a memory pool differs slightly depending on whether you intend the memory pool to be used by a network protocol, such as IPv4, or an END driver.

All memory pools are organized around pools of tuples. However, because a network protocol typically requires clusters of several different sizes, its memory pool must contain several tuple pools (one tuple pool for each cluster size). In addition, each cluster size *must* be a power of two. Common cluster sizes for this style of memory pool are 64, 128, 256, 512, 1024, and 2048 bytes. See Figure 3-4.

Figure 3-4 **A Protocol Memory Pool**




By contrast, a memory pool intended for an END driver typically uses only one cluster size and the cluster size is not limited to a power of two. Thus, you are free to choose whatever cluster size is most convenient, which is typically something close to the maximum transmission unit (MTU) of the network. A network's MTU is typically 1500 bytes.

For more information on memory pools, see the reference entry for **netBufLib**.

### Establishing a Network Driver Pool

The following steps illustrate how to use **netPoolCreate( )** with **_pLinkPoolFuncTbl** to establish a network driver pool:

1. Allocate memory for a network buffer configuration structure and add enough space to also hold 8 additional bytes for the **pDrvCtrl->pNetBufCfg->pName** field.

```
if (pDrvCtrl->pNetBufCfg = (NETBUF_CFG *) memalign (sizeof(long),
                                                    (sizeof(NETBUF_CFG) +
                                                     END_NAME_MAX)) == NULL)
    return (ERROR);

bzero(pDrvCtrl->pNetBufCfg,sizeof(NETBUF_CFG));
```

2. Initialize the **pName** field.

```
pDrvCtrl->pNetBufCfg->pName = (char *)((int)pDrvCtrl->pNetBufCfg +
                                       sizeof(NETBUF_CFG));

sprintf(pDrvCtrl->pNetBufCfg->pName,"%s%d","fei", pDrvCtrl->unit);
```

3. Set the attributes to be cached, cache-aligned, sharable, and ISR safe.

```
pDrvCtrl->pNetBufCfg->attributes = ATTR_AC_SH_ISR;
```

4. Use a **NULL** value to set **pDomain** to kernel. This instructs **netPoolCreate( )** to allocate memory accessible in the kernel domain.

```
pDrvCtrl->pNetBufCfg->pDomain = NULL;
```

5. Set the ratio of **mBlk**s to clusters.

```
pDrvCtrl->pNetBufCfg->ctrlNumber = pDrvCtrl->nClusters * 10;
```

6. Use a **NULL** value to set the memory partition of **mBlk**s to kernel.

```
pDrvCtrl->pNetBufCfg->ctrlPartId = NULL;
```

7. For now, set extra memory size to zero.

```
pDrvCtrl->pNetBufCfg->bMemExtraSize = 0;
```

8. Set the cluster memory partition to kernel, use **NULL**.

```
pDrvCtrl->pNetBufCfg->bMemPartId = NULL;
```

9. Allocate memory for the network cluster descriptor.

```
pDrvCtrl->pNetBufCfg->pClDescTbl =(NETBUF_CL_DESC *)memalign
                                  (sizeof(long), sizeof(NETBUF_CL_DESC));
```

10. Initialize the cluster descriptor.

```
pDrvCtrl->pNetBufCfg->pClDescTbl->clSize = CLUSTER_SIZE;
pDrvCtrl->pNetBufCfg->pClDescTbl->clNum = pDrvCtrl->nClusters * 10;
pDrvCtrl->pNetBufCfg->clDescTblNumEnt = 1;
```

11. Call **netPoolCreate( )** with the link pool function table.

```
if ((pDrvCtrl->endObj.pNetPool =
        netPoolCreate ((NETBUF_CFG *)pDrvCtrl->pNetBufCfg,
            _pLinkPoolFuncTbl)) == NULL)
    return (ERROR);
```

12. Free the **pDrvCtrl->pNetBufCfg** and **pDrvCtrl->pNetBufCfg->pCIDescTbl**.

```
free (pDrvCtrl->pNetBufCfg->pClDescTbl);
free (pDrvCtrl->pNetBufCfg);
```

### Handling Packet Reception

The list of END driver entry points (see Table 3-2) makes no mention of an *end***Receive( )** entry point. That is because an END driver does not require one. Of course, your driver must include code that handles packet reception, but the MUX never calls this code directly. Thus, the specifics of the code for packet reception are left to you.

However, even if the MUX API does not require an *end***Receive( )** entry point, you need to consider the VxWorks system when designing your driver's packet reception code. For example, your network interface driver must include an entry point that acts as your device's interrupt service routine. In addition, your driver also needs a different entry point for completing packet reception at the task level.

Internally, your task-level packet-reception entry point should do whatever is necessary to prepare the packet for handing off to the MUX, such as ensuring data coherency. Likewise, this entry point might use a level of indirection in order to check for and avoid race conditions before it attempts to do any processing on the received data. When all is ready, your driver should pass the packet up to the MUX. To do this, it calls the routine referenced in the **receiveRtn** member of the **END_OBJ** structure (see *Providing Network Device Abstraction: END_OBJ*, p.58).

Although your driver's *end***Load( )** entry point allocated this **END_OBJ** structure and set the values of most of its members, it did not and could not set the value of the **receiveRtn** member. The MUX does this for you upon completion of the **muxDevLoad( )** call that loads your driver. However, there is a very brief interval between the time the driver becomes active and the completion of **muxDevLoad( )**. During that time, **receiveRtn** is not set. Thus, it is always good practice to check **receiveRtn** for **NULL** before you try to execute the routine referenced there.

#### Receive Handler

A network device is initialized with the base pointer to a ring of descriptors. The device uses these descriptors to:

- locate a buffer into which it can write incoming data
- communicate status to the device driver

The device cycles through the descriptor ring. When the device receives an incoming Ethernet frame, it receives it into its FIFO. The device then writes the frame into the buffer which it locates through the currently accessed descriptor. The prevalent method used for a device to write data into both the descriptors and the buffers, is direct memory access (DMA).

As the network device indexes around the descriptor ring, it tests each entry for availability. When the device receives a frame and finds an available descriptor, its DMA engine fills the associated buffer and sets a status flag in the descriptor indicating that the buffer is full.

If a device encounters a used descriptor or an end-of-ring marker, the device halts and enters a stalled state. The stalled state means that the device has lapped the device driver's ring servicing. Minimally, the device driver must then clear the next descriptor the device has on its list. Some devices may require the driver software to move the end-of-ring marker and possibly restart the receiver.

A driver's receive handler is responsible for navigating the device's descriptor ring, determining which descriptors are filled, and then passing the buffers up to the network stack. After the receive handler has given a descriptor's filled buffer to the stack, it clears the descriptor and replenishes it with a new buffer. To be efficient, the receive handler must continue to handle descriptors as long as it detects that completed DMA transfers have occurred. However, there is no guarantee that the handler will ever become idle. When writing a device driver, you must assume that the rest of the operating system requires time for its own tasks, and that other END drivers using the **tNetTask** context require CPU time to function. So, care must be taken to prevent a single driver from monopolizing either the CPU or **tNetTask**.

The example receive handler described in this document has the following features:

- A Receive Loop—A while loop predicated on testing successive descriptors arranged in a ring. This loop continues to run as long as the descriptors indicate there is additional work available.
- Fair Access Bounding—A limit to how long a receive handler continuously services its descriptors before relinquishing the operation so another device can service its descriptors.
- Receive Handler Interlocking Flag—A lightweight semaphore to protect against redundant scheduling of a receive handler.
- Receiver Stall Handling—An action to restart a device’s receiver if it has suffered a stall. The action is only necessary if the device halts on a receive stall and requires a register state to be cleared.
- Interrupt Re-Enabling—Setup for resumption of operation at an undetermined future time.
- Two-Tiered Polling—A rescheduling scheme that allows for a reduction in interrupt load.

### Receive Loop

An efficient receive loop is vital to a high performance END driver. It is imperative to do only what is absolutely necessary in the loop itself. Any extraneous code within this loop has a negative performance impact. Great care must be taken to stage as much as is possible outside the loop. If a decision or calculation can possibly be made during initialization, every effort should be expended to do so. Complexity of initialization is a one time cost, whereas any work done in the loop is repeated an enormous number of times.

The receive-loop's function is to service the receiver's DMA ring. This entails:

- Determining which descriptors have buffers that hold completed DMA transfers
- Determining whether incoming frames are to be handled or discarded
- Retrieving and replacing DMA buffers
- Ensuring cache coherency of DMA buffers
- Passing properly configured tuples up the stack
- Returning used descriptors to an available state
- Bounding, to avoid monopolizing the CPU or network stack

#### Efficient Receive Loop

The receive-loop traverses the receive-ring and reads the status of each descriptor it encounters. An efficient receive-loop should make use of the fact that the memory for the descriptors is allocated in a single contiguous block. This allows the descriptor ring to be accessed as an array regardless of the method the device uses to traverse the ring. Arrays are much faster than linked lists. Because an array is always a block of contiguous memory, a compiler can optimize array accesses for certain considerations, such as caching and fetching. For example, the compiler knows that if the base address of an array is cached, the remainder of the array is cached as well (for smaller arrays). On the contrary, if the first address in a linked list is cached, the compiler cannot assume that the next address in the linked list is also cached. When the compiler accesses a new item in the array, it must only add an offset to find the new item. If code is traversing a linked list, then the compiler must fetch the base address for each node in the linked list. Because fetches are generally slower than the arithmetic of adding an offset, the array—which replaces fetches with the offset addition—runs much faster.

```
while((rbdStatus)
```

#### Obtaining a New Tuple

Within the while-loop, it must be determined whether incoming frames can be handled or must be discarded. The receive-loop can only handle those frames for

which it can obtain resources. These resources are obtained from the net pool with **netTupleGet( )**. If **netTupleGet( )** returns a **NULL**, meaning that there are no resources available, the receive-loop must discard that frame. The receive handler has the option to break out of the loop and return later, when resources may again be available, or to continue traversing the ring, discarding outstanding frames.

```
if ((pNewMblk = netTupleGet(pDrvCtrl>endObj.pNetPool, CLUSTER_SIZE,
                            M_DONTWAIT, MT_DATA,0)) == NULL)
    {
    endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_IN_ERROR);
    endM2Packet (&pDrvCtrl->endObj, NULL, M@_PACKET_IN_DISCARD);

    return (ERROR);
    }
```

#### Retrieving and Replacing DMA Buffers

To receive and replace DMA buffers, you use the following code sequence:

1. Retrieve the used tuple as follows:

   ```
   pMblk = pDrvCtrl->pMblkList[pDrvCtrl->index];
   ```

2. Place a new tuple on the association list:

   ```
   pDrvCtrl->pMblkList[pDrvCtrl->index] = pNewMblk;
   ```

   If the device supports DMA to a 2 byte offset, move the **mBlk** data pointer by 2 bytes:

   ```
   pNewMblk->mBlkHdr.mData = pNewMblk->pClBlk->clNode.pClBuf +
                             pDrvCtrl->offset;
   ```

3. Ensure cache coherency of the DMA buffers as follows:

   ```
   DRV_CACHE_INVALIDATE (pNewMblk->pClBlk->clNode.pClBuf,
                         CLUSTER_SIZE);
   ```

4. Convert the buffer virtual address to a physical address:

   ```
   pBuffer = VIRT_TO_PHYS ((UINT32) pNewMblk->mBlkHdr.mData;
   ```

5. Update the receive descriptor:

   ```
   xxxDescBufWrite ((&pDrvCtrl->pRxDescBase[pDrvCtrl->index], pBuffer,
                     BUFFER_OFFSET);
   ```

6. Copy DMA length to **mBlk**:

   ```
   pMblk->mBlkHdr.mLen =
         (xxxDescRead (&pDrvCtrl>pRxDescBase[pDrvCtrl>index]) & ~0xc000);
   ```

#### Clearing the Descriptor Status

You can clear the descriptor status by using the following code:

```
xxxDescStatusClear (&pDrvCtrl->pRxDescBase[pDrvCtrl->index])
```

#### Incrementing the index

Next, increment the index:

```
pDrvCtrl->index = (++pDrvCtrl->index % pDrvCtrl->rbdNum);
```

#### Sending a Received Frame to the Stack:

To pass a buffer up to the MUX, a driver calls **muxReceive( )**, which in turn calls the protocol's **stackRcvRtn( )** routine (see *Passing a Packet Up to the Protocol: stackRcvRtn( )*, p.19). When control returns from **muxReceive( )**, the driver can consider the data delivered and can forget about the buffers it handed up to the MUX. When the upper layers are done with the data, they free the buffers back to the driver's memory pool. The macro, **END_RCV_RTN_CALL**, which is provided by Wind River, calls **muxReceive( )**.

```
END_RCV_RTN_CALL (&pDrvCtrl->endObj, pRbdTag->pMblk);

endM2Packet (&pDrvCtrl->endObj, pRbdTag->pMblk, M2_PACKET_IN);
```

### Fair Access Bounding

In a polling architecture, it is possible for a single device to be receiving a continuous stream of frames. In this case, the device's device driver receive handler could possibly starve other device's drivers, or even the whole system, for CPU cycles. Therefore, it is necessary to employ *fair access bounding* to avoid a single device's receive handler monopolizing the CPU.

The technique for fair access bounding is to simply set a policy of how many frames a receive handler is allowed to service before relinquishing operation. Then, when the receive handler has serviced that number of frames, the receive handlers' current execution is terminated and rescheduled, if necessary. The determination of whether the receive handler needs to be rescheduled is based on whether or not there were additional received frames outstanding. This is determined by testing the next descriptor to be serviced. If the descriptor indicates a received frame (full descriptor), the receive handler must be rescheduled. If the descriptor indicates that there are no outstanding frames (empty descriptors) then the receive handler re-enables the receive interrupt and exits.

```
int loopcounter = pDrvCtrl->maxRxFrames;   /* local variable */
                                           /* in receive handler */
```

```
while ((rbdStatus != RBD_STATUS_FREE) && (--loopcounter > 0))
    {
    /* Receive Loop */
    }

if (rbdStatus != RBD_STATUS_FREE)
    {
    /* Put this job back on the netJobRing and leave */

if ((netJobAdd ((FUNCPTR) xxxRecvHandler, (int) pDrvCtrl,
                    0,0,0,0)) == ERROR)
        {
        /* Very bad!! The stack is now probably corrupt. */
        logMsg("The netJobRing is full. 2\n",0,0,0,0,0,0);
        return;
        }
    }
else
    {
    pDrvCtrl->rxJobQued = FALSE;
    }
```

### Receive Handler Interlocking Flag

VxWorks limits the work that can be done in an ISR. Because of this limitation, much of the work related to servicing interrupt conditions must be deferred outside of an ISR to other code executing in a task level context. Any program, such as a device driver, that deals with hardware interrupts must inevitably defer to a substantial amount of work that arises from servicing ISRs. To accommodate deferring work from ISRs, Wind River's network stack provides the scheduling utility **netJobAdd( )**, which operates in the **tNetTask** context. **netJobAdd( )** uses a facility called the **netJobRing**. The **netJobRing** is used by both the device driver and by the network stack. This facility is a limited resource so you must take great care when writing your device driver to safeguard against overflowing this ring. If the ring is allowed to overflow, the state of the network stack can be corrupted.

The limitations imposed on interrupts by VxWorks are primarily due to the systemic impact that interrupts impose. Although there is no expectation of determinism associated with END drivers, or with the network stack, there is also a mandate that they not interfere with the ability of other programs operating in the same environment to archive determinism. In addition, interrupts impose context switch overhead and have a tendency to reduce efficiency for many architectures. Because interrupts are relatively costly in terms of overall system performance, one goal of an END driver is to prevent interrupts to be generated.

The work most often done by an interrupt's task-level service routine involves servicing a queue. It is efficient to continue to service this queue for as long as work is available. Because service routines continue to execute as long as there is work

to do, scheduling another instance of a service routine while one is already running is unnecessary and redundant. Because of this, you should try to coalesce interrupts. This can be accomplished in software by masking an incident interrupt in its ISR and leaving that interrupt masked while the service routine is running. Then, before exiting the service routine, re-enable the interrupt.

Because of the complexities associated with the physical arrangement and logical handling of interrupts, simply masking interrupts is an inadequate solution. It is often the case where several discrete devices share the same physical interrupt line. The logical organization is that the ISRs for each discrete device on that same interrupt line are daisy chained in a linked list. When one of the devices on the interrupt line generates an interrupt, the system interrupt logic walks down the daisy chain calling each ISR in turn. Besides wasting CPU cycles, this procedure also has a dangerous side effect. As discussed previously, END drivers mask a particular interrupt when its ISR is executed. Unfortunately, this does not mean that the interrupt bit in a device's status register fails to be set for subsequent occurrences of the same kind of event. It only guarantees that the device will not generate another interrupt of the same type as the one that is masked. If another device on the same interrupt line generates an interrupt, the ISR for the network device executes and tests the device's status register. If another event of a given type has occurred since the interrupt for that type of event was masked, the ISR still detects that the device has an interrupt bit set. If this occurs, the ISR erroneously schedules a task-level service routine on the **netJobRing**, even though it masked the device's interrupt to prevent this from happening. This phenomenon occurs in some systems with enormous frequency and with catastrophic effect due to overflow of the **netJobRing**.

To safeguard against redundant scheduling of task level service routines, you must employ additional means of protection for **netJobRing**. The mechanism to do this appears to be a semaphore. However, a semaphore may be too heavy for this particular application because it has more overhead than is justified by the problem and it would need to execute in a particularly performance sensitive location. A lighter means of providing protection is a simple boolean flag. A *receive handler interlocking flag* is a device instance-specific flag that is kept in the driver's **DRV_CTRL** structure. The END driver's ISR checks this flag before scheduling the associated service routine on **netJobRing**. If the flag is not set, the ISR schedules the service routine and sets the flag. If the flag is already set, the ISR skips scheduling the routine. The flag is cleared in the service routine after it completes execution. Using a flag in this manner introduces the possibility of a race condition. However, the risk associated with the race condition is insignificant. If an occasional case of redundant scheduling occurs, it is unlikely to cause any problem. It is also true that if, on occasion, a service routine is slightly delayed from

getting scheduled on the **netJobRing,** any subsequent delay in receiving a small number of packets is easily tolerated by the network stack.

#### Implementing Receive Handler Interlocking Flag

1. Add the receive handler interlock flag to the **DRV_CTRL** structure as follows:

```
BOOL        rxJobQued;    /* fei82557RecvHandler() queuing flag */
```

2. In the device driver's receive ISR, test the receive handler interlock flag prior to calling **netJobAdd( )** and schedule the receive handler service routine:

```
/* Test if fei82557RecvHandler() is on netJobRing. */

if(!pDrvCtrl->rxJobQued)
    {
    /* fei82557RecvHandler() is not on netJobRing so put it on. */

    if ((netJobAdd ((FUNCPTR) fei82557RecvHandler, (int) pDrvCtrl,
                0, 0, 0, 0)) != ERROR)
        {
        pDrvCtrl->rxJobQued = TRUE;
        }
    else
        {
        logMsg("The netJobRing is full. 1\n",0,0,0,0,0,0);

        I82557_INT_ENABLE(SCB_C_M);
        return;
        }
    }
```

3. At the end of the receive handler service routine, after it is certain that the routine has completed execution and reschedules itself, clear the receive handler interlock flag by setting it to **FALSE**.

```
pDrvCtrl->rxJobQued = FALSE;
```

### Receiver Stall Handling

As discussed previously, a stall condition occurs when a device driver allows a device to temporarily exhaust its available resources. In the case of a receive stall, the device has lapped the receive descriptor ring and has no available buffers into which it can direct DMAs. Devices typically behave in one of two ways when this occurs:

- Some devices simply require that the next descriptor in the sequence be cleared. That is, the descriptor's status must be set to free or available. In this case, the device automatically detects that the stall is cleared and resumes operation without any action on the part of the driver.

- Other devices place their receiver into a halted state by setting a bit in a control register. For this type of device, it is often required that, in addition to freeing the next descriptor, the driver must clear the control register bit before operation resumes.



#### Interrupt Re-Enabling

END drivers mask interrupts in the ISR before scheduling a service routine. The nature of the work done by these service routines is to repetitively service one item after another from a queue. The service routine continues to service items as long as it determines there is more work to be done. It is unnecessary and detrimental to performance to allow additional interrupts to schedule service routines for work that is already being done by a previously scheduled run of the service routine. Hence, it is general practice to mask interrupts in ISRs. In the case of the receive interrupt, the scheduled service routine is the device driver's receive handler.

As discussed previously, masking an interrupt in a device does not guarantee that the device will not record the event in a status register. It only implies that the device does not actually generate the interrupt. In addition to recording events in a status register while an interrupt remains masked, some devices immediately generate an interrupt when the mask is cleared if events occurred while the interrupt was masked. In the case of an END driver, and in the receive handler in particular, the events that caused the status bit to be set would have already been serviced by the service routine. The device driver writer should note if the device for which the driver is being written exhibits this characteristic. If so, care should be taken to clear the event before unmasking the receive interrupt mask.

In all cases, as with the receive handler interlocking flag, the receive interrupt should only be unmasked when it is certain that the receive handler has completed execution.

#### Two-Tiered Polling

The technique previously used for scheduling a receive handler in END drivers involved a single tier polling method, referred to as *interrupt stimulated polling*. Using the interrupt stimulated polling method, the device would receive an incoming packet into its DMA ring and generate an interrupt. The interrupt handler would then disable the device's interrupt and schedule a receive handler to run **tNetTask**. This receive handler then polled all descriptors on the DMA ring for the original packet that caused the interrupt and any additional DMAs that occurred since that initial DMA. When the receive handler finished servicing all of the completed DMAs, it would re-enable the device's interrupt and exit.

The intention of this method was to service the maximum number of packets possible for each interrupt. This method attempted to relieve the system of the overhead implicit with frequent interrupts. However, interrupt stimulated polling resulted in an interrupt occurring for almost all received packets. This imposed considerable overhead on the system when servicing the large number of interrupts associated with high traffic loads.

The interrupt stimulated polling method fails because devices do not update descriptors until after DMAs are complete. Therefore, there is a race condition between the service of the previous packet and the ongoing reception and DMA of the next. If the service of the first packet completes before the next packet's DMA completes, the check of the next packet's descriptor does not indicate an ongoing DMA. When this occurs, the receive handler terminates the polling, re-enables the device's interrupt, and exits. The receive handler then misses an additional incoming packet whose DMA is not yet complete.

The outcome of this is that the next received packet also generates an interrupt. The timing is such that if the CPU executes a single pass of the receive handler in less time than a subsequent reception and DMA, which is a fixed time depending on the network bit rate, the network interface generates a large quantity of interrupts. This gives the appearance of an interrupt driven mechanism when it is in fact interrupt stimulated polling.

This problem is currently prevalent with 100 Mb networks. However, as CPU speeds increase and network bit rates are fixed at specific stops, it is only a matter of time before this phenomenon becomes prevalent with faster bit rates as well.

#### Explanation of the Two Tiered Polling Method With Fair Access Bounding.

Two-tiered polling is a polling method consisting of an inner and an outer loop of polling. The two-tiered polling method is initiated, like the interrupt stimulated method, by an initial packet causing the device to generate an interrupt. However, the two-tiered polling method continues to poll for additional incoming packets for a specified number of times.

At the heart of two-tiered polling are the controlling variables:

**pollDone**
: A flag indicating whether the outer loop continues polling.

**pollCnt**
: A counter tracking successive times a receive handler encountered a descriptor indicating it does not need to be serviced (an empty descriptor).

**pollLoops**
: The maximum times the outer loop can increment before terminating.

#### Operation Details

After a receive handler that has been scheduled to run by the receive interrupt handler begins execution:

1. The Receive Handler obtains and tests the next descriptor to be serviced
   a. If the next descriptor indicates it needs service (full), the receive handler enters the receive loop and the counter, **pollCnt**, is cleared
   b. If the next descriptor is empty, the receive handler exits without changing the status, counters, the index, or pointers.
2. When the receive handler enters the receive loop, the receive loop continues to service its descriptors, until it encounters one of two conditions:
   - It encounters an empty descriptor.
   - It reaches the maximum packet boundary set by the fair access limit.
3. After exiting the receive loop, the receive handler tests if either of two conditions exist:
   - The counter, **pollCnt**, is less than the value of **pollLoops**.
   - The next descriptor indicates it needs to be serviced (full).

   If either condition is true, the receive handler's behavior depends on the status of the next descriptor to be serviced.
   a. If the next descriptor to be serviced is empty, the while loop must have terminated because it encountered an empty descriptor. The following actions are taken:
      i. The receive handler increments **pollCnt**.
      ii. The receive handler places itself back on the **netJobRing** to be executed again.
      iii. The receive handler sets the **pollDone** flag to **FALSE** indicating a continuation of the outer loop of polling.
   b. If the status of the next descriptor is full, the receive loop must have terminated because it reached the maximum number of descriptors to be serviced before relinquishing operation. The following actions are taken:
      i. The receive handler clears **pollCnt**.
      ii. The receive handler then places itself back on the **netJobRing** to be executed again.

iii. The receive handler sets the **pollDone** flag to **FALSE** indicating that it will continue the outer tier of polling.

If neither of the conditions are true, the receive handler terminates the outer loop of polling and the following actions are taken:

i. The receive handler clears the receive handler interlock flag.

ii. The receive handler clears **pollCnt**.

iii. The receive handler sets **pollDone** to **TRUE**.

4. Before it exits, the receive handler tests **pollDone**. If **pollDone** is **TRUE**, the receive handler re-enables the device's receive interrupt.
5. The receive handler exits.

#### How to Implement Two-Tiered Polling With Fair Access Bounding

1. Add two-tiered polling fields to the **DRV_CTRL** structure.

```
BOOL        pollDone;    /* Flag indicating outer loop exit */
UINT32      pollCnt;     /* polling counter */
UINT32      pollLoops;   /* polling limit */
```

2. Add the fair access limitation field to the **DRV_CTRL** structure.

```
UINT        maxRxFrames; /* max frames to Receive in one job */
```

3. In the driver's *end***Load( )** routine, specify the addition of the **maxRxFrames** parameter to the **END_LOAD_STRING**.

```
/*
 * The <maxRxFrames> parameter limits the number of frames the
 * receive handler services in one pass. It is intended to
 * prevent the tNetTask from monopolizing the CPU and starving
 * applications. This parameter is optional, the default value
 * is nRFDs * 2.
 */
```

4. In the driver's parsing routine, add an optional parse for the **maxRxFrames** parameter.

```
/* passing maxRxFrames is optional. The default is 128  */

pDrvCtrl->maxRxFrames = pDrvCtrl->nRFDs * 2;
tok = strtok_r (NULL, ":", &holder);

if ((tok != NULL) && (tok != (char *)-1))
    pDrvCtrl->maxRxFrames = strtoul (tok, NULL, 16);
```

5. In the driver's start routine, initialize the two-tiered polling fields in the **DRV_CTRL** structure.

```
pDrvCtrl->pollCnt = 0;
pDrvCtrl->pollLoops = 1;
pDrvCtrl->pollDone = FALSE;
```

6. In the task-level receive handler, add a local variable to use as a loop counter. This is used to bound the maximum number of packets that can be serviced for a single pass through the handler.

```
int   loopCounter = pDrvCtrl->maxRxFrames;
```

7. In the receive handler, terminate the receive loop **while** loop by decrementing the local variable **loopCounter**.

```
while((rbdStatus != RBD_STATUS_FREE) && (--loopCounter > 0))
```

8. In the receive handler, immediately after the end of the receive loop, add the two-tiered polling code.

```
if ((pDrvCtrl->pollCnt < pDrvCtrl->pollLoops) ||
    (rbdStatus != RBD_STATUS_FREE))
    {
    if (rbdStatus == RBD_STATUS_FREE)
        pDrvCtrl->pollCnt++;
    else
        pDrvCtrl->pollCnt = 0;

    pDrvCtrl->pollDone = FALSE;

    /* Put this job back on the netJobRing and leave */

    if ((netJobAdd ((FUNCPTR) fei82557RecvHandler, (int) pDrvCtrl,
                    0,0,0,0)) == ERROR)
        {
        /* Very bad!! The stack is now probably corrupt. */

        logMsg("The netJobRing is full. 2\n",0,0,0,0,0,0);

        I82557_INT_ENABLE(SCB_C_M);
        return;
        }
    }
else
    {
    pDrvCtrl->pollCnt = 0;
    pDrvCtrl->pollDone = TRUE;
    pDrvCtrl->rxJobQued = FALSE;
    }
```

9. Immediately before leaving the task level receive handler, re-enable the device's receive interrupt (only if polling is done).

```
if (pDrvCtrl->pollDone)
    {
    I82557_INT_ENABLE(SCB_C_M);
    }
```

### Handling Packet Transmission

Unlike the receive handler, the driver's *end***Send( )** routine is called from multiple contexts—network applications or **tNetTask**—which may supersede each other. The send routine also manipulates linked lists which must be protected from corruption. Care must be taken to safeguard the send routine from concurrent access. Therefore, the *end***Send( )** routine must always take the transmit semaphore stored in **END_OBJ**, by calling **END_TX_SEM_TAKE( )**.

#### Transmit-Packet-Complete Handler Interlocking Flag

Transmit-packet-complete interrupts are typically used to allow the driver to return resources to the pool after a packet is transmitted. The frequency of these interrupts can be very high. Because of the high frequency at which these interrupts are generated, transmit-packet-complete interrupts can degrade system performance and overflow **netJobRing**. *Transmit Descriptor Clean-up*, p.54 includes a discussion of how to reduce the frequency of this interrupt. This section deals with how to prevent the transmit-packet-complete interrupt from causing a **netJobRing** overflow. The method used is essentially the same as that used for the receive handler interlocking flag (see *Receive Handler Interlocking Flag*, p.44).

A transmit-packet-complete handler interlocking flag is a device instance-specific flag that is kept in the driver's **DRV_CTRL** structure. The END driver's ISR checks this flag before scheduling the associated service routine on **netJobRing**. If the flag is not set, the ISR schedules the service routine and sets the flag. If the flag is already set, the ISR does not schedule the routine. The flag is cleared in the service routine after it completes execution.

#### Supporting Scatter-Gather

Scatter-gather is a DMA technique that allows for a single large block of data to be distributed—or *scattered*—among multiple buffers. The data can then be *gathered* together later and transferred in a single DMA transaction, as if it were stored in a contiguous buffer. This capability is desirable because the network stack is often unable to find a single cluster buffer that is large enough to hold a large packet. That is, when the network is unable to find a buffer of sufficient size, it must obtain multiple tuples with cluster buffers that, cumulatively, have sufficient space to hold the packet. The stack then fragments the packet among multiple tuples. For transmit, the fragmented packet is sent as an **mBlk** chain to the driver's send routine to be transmitted.

**NOTE:** In END drivers, scatter-gather is not a concern for packet reception. This is because the driver's buffers are all of a single size and are sufficient to hold the maximum incoming frame (MTU). Therefore, END drivers do not fragment incoming frames.



When scatter-gather is not supported by the device and the driver is sent a fragmented packet, the driver must obtain a single buffer from its pool and must then copy the packet fragments into a single buffer. This is possible because the driver pool, unlike the network stack pool, typically has only a single buffer size that is sufficient to hold the largest packet the maximum transfer unit (MTU) allows. This means that in most cases, the driver can find a buffer that is large enough to accommodate any packet. However, the overhead of requiring the driver to obtain a buffer and copy the packet fragments into the buffer is a substantial drag on overall system performance.

When a device supports scatter-gather, it can continue DMA across multiple fragments by following a list of fragment buffer pointer and size pairs. A driver written for such a device walks the **mBlk** chain, extracts the cluster buffer pointers and the fragment sizes, and then forms a gather list according to the device's specification.

Devices typically use one of two common mechanisms for creating gather lists. The first method requires the device to read the buffer pointer and size pairs out of a list contained in a single transmit descriptor. The second mechanism requires the device to follow a list of descriptors that are tied together, reading in turn the successive buffer pointer and size pairs from each descriptor in the list.

The driver's send routine is responsible for determining if the driver has sufficient resources to handle an outgoing packet. Once the send routine has made this determination, the routine is responsible for taking the appropriate action.

To determine whether or not there are sufficient resources available to hold the packet data, a send routine must count the number of fragments in the **mBlk** chain, and compare that number with the amount of resources the driver currently has available. Determining the amount of resources available depends on the device's gather mechanism. As described previously, devices typically employ one of two common gather mechanisms. (There is also a hybrid method that uses multiple pairs across multiple descriptors, but this type is rarely used and it is usually the case that if a descriptor holds multiple pointer and size pairs, the entire packet must be held by a single descriptor's pair list.) In all of these methods, the problem for the driver is to determine the number of fragment pairs that can be held by the descriptors that are currently free.

If the number of available descriptors is insufficient to hold the packet data, the send routine attempts to free enough descriptors to handle the packet. If the send routine fails to free a sufficient number of descriptors, it must then either coalesce the packet into a single buffer—the same practice that is used if scatter-gather is not supported—or it must throw the packet away.

If the send routine determines that it *does* have sufficient resources to handle the outgoing packet, the driver must then walk the **mBlk** chain. For each tuple in the chain, the driver must write the cluster buffer pointer into a free descriptor's buffer pointer field or list, and then attach the free descriptors it is using together into a list to be placed on the transmit queue. While the fragment pointers are being transferred to the descriptor(s), the descriptor fields should be updated to reflect that they hold buffer pointers that are ready for transmit. If the device specifies that fragments be distributed over a list of descriptors, the device also specifies that the first and last descriptors in the list be marked accordingly. After the fragment pointers and sizes for the packet's entire **mBlk** chain have been transferred to the descriptor list and the descriptor fields are set up in the manner expected by the device, the assembled list is placed at the end of the transmit queue.

#### Transmit Descriptor Clean-up

The driver's send routine is also responsible for storing the **mBlk** pointer to the **mBlk** chain holding the packet in such a way that it can be later correlated to the associated descriptor or descriptors on the transmit queue.

After a packet is successfully transmitted, most devices generate a packet-complete interrupt. The ISR for this interrupt causes the driver's *transmit-packet-complete handler* to be scheduled, which in turn calls the driver's *transmit descriptor clean* routine to free the packet descriptor or descriptors and the associated **mBlk** chain. As described in *Transmit-Packet-Complete Handler Interlocking Flag*, p.52, numerous packet-complete interrupts are a detriment to performance.

The driver's send routine may also directly call the transmit descriptor clean routine. This can be a highly effective method for initiating transmit descriptor cleanup. However, there are two issues that should be considered:

- When the send routine calls the transmit descriptor clean routine, the device may not have actually transmitted the packet and there may be little or nothing to clean. Therefore, the descriptor cleanup often depends on subsequent calls to the send routine to clean up previously used descriptors.
- Calling the transmit descriptor clean routine for every packet sent imposes substantial overhead.

In some circumstances, the first consideration can result in a transmit stall or even deadlock. The solution to this transmit stall is to continue to allow the packet-complete interrupt to occur but control the frequency at which it is generated. This gives a backup to the send routine's cleanup attempts.

To control the frequency of the packet-complete interrupt, keep it masked, and only unmask it when a call to the transmit descriptor clean routine fails to free sufficient descriptors.

To determine if sufficient descriptors have been freed:

- Establish a threshold of some percentage of the transmit descriptors
- If the send routine's call to the transmit descriptor clean routine does not increase the free count to greater than the threshold amount, unmask the packet-complete interrupt

The solution to the transmit descriptor clean overhead is to once again track the free transmit descriptor count and to only call the transmit descriptor clean routine when the free count falls below a certain threshold.

Now put these two mitigators together:

- Only call the transmit descriptor clean routine when the free transmit descriptor count falls below a certain threshold.
- If the send routine's call to the transmit descriptor clean routine does not increase the free count to a value greater than the given threshold, unmask the packet-complete interrupt.

#### Transmit Descriptor Indexing

The memory for the driver's transmit descriptors should be contiguously allocated. This allows the driver's send routine to access the descriptors with an index from the base pointer returned by the allocation. This is similar to the indexing scheme used by the receive handler routine. Like the receive handler routine, the driver's send routine should treat the transmit descriptors as a circular array, or a *transmit descriptor ring*.

One of the issues that the driver's send routine must address is that it must track the transmit descriptors on two different queues, the *free queue* and the *used queue*. These queues are defined as follows:

free queue
: Lists descriptors currently available for use.

used queue
: Lists descriptors currently on the transmit queue.

These queues are actually different dynamic parts of the same list of descriptors. Setting up and efficiently managing these queues is a critical part of a send routine's design. To manage these queues the driver establishes two indices, one for each queue.

The index for the free queue—the *free index*—references the next available descriptor available for use by the driver's send routine. The send routine should follow the free index around the transmit descriptor ring. When the send routine places a descriptor on the device's transmit queue, it increments the free index. In order to track how many descriptors are currently free, the send routine also decrements a *free counter*. The initial state for the free counter is the total number of transmit-descriptors allocated to the driver.

The index for the used queue references the descriptor that has been on the device's transmit queue for the longest period of time. The used queue is also the *next-to-clean queue*. The index for the next-to-clean queue is the *clean index*, this references the next transmit descriptor to be cleaned.

#### Transmit Packet Association List

As stated previously, it is the responsibility of the driver's send to store a transmitted packet's **mBlk** chain pointer in such a way that it can be later correlated to the associated descriptor or descriptors on the transmit queue. The mechanism to do this is a *transmit packet association list*.

This list is an array of **mBlk** pointers that is of equal length to the total number of transmit descriptors allocated by the driver. This list is accessed using the same indices that the driver uses to reference the descriptors. When the send routine places a descriptor on the device transmit queue, it uses the free index to correlate the transmit packet association list to the transmit descriptor ring. As the send routine moves around the transmit descriptor ring, for each fragment buffer pointer it puts into a descriptor, it determines if that fragment is the last fragment for the packet it is transmitting. If it is the last fragment for the packet, the send routine puts the pointer to the packet's **mBlk** chain into the transmit packet association list at the same index as the descriptor that holds the packet's last fragment. If the fragment is not the last fragment of packet, the send routine sets the correlating transmit packet association list entry to **NULL**.

#### Transmit-Packet-Complete Handler

The transmit-packet-complete handler is a task-level routine that is scheduled by the transmit-packet-complete interrupt's ISR. This interrupt occurs when the device has completed transmitting a packet. It is used to indicate to the driver that it can now clean the transmit descriptors used for the transmission of that packet.

When the transmit-packet-complete interrupt's ISR executes, it masks the transmit-packet-complete interrupt.

The transmit-packet-complete handler must guarantee that the driver's transmit descriptor clean routine is called in a safe manner. This is a requirement because the transmit descriptor clean routine manipulates the device's transmit queue. Because the device's transmit queue is asynchronously accessed by multiple contexts, it must be protected by a mutual exclusion semaphore. Therefore, the transmit-packet-complete handler must take the driver's transmit semaphore before calling the transmit descriptor clean routine. It must also immediately give the semaphore after the transmit descriptor clean routine returns.

The transmit-packet-complete handler must guarantee that a minimum amount of transmit descriptors are freed before it stops. To this goal, it tests that the call to transmit descriptor clean increases the free count to the required threshold.

- If the free count is less than the threshold, the transmit-packet-complete handler reschedules itself, and leaves the transmit-packet-complete interrupt masked and the transmit-packet-complete handler interlock flag set.
- If the free count is increased to greater than or equal to the threshold, the transmit-packet-complete handler clears the transmit-packet-complete interrupt mask, clears the transmit-packet-complete handler interlock flag, and exits.

#### Transmit Descriptor Clean

The transmit descriptor clean routine is responsible for returning transmit descriptors back to a usable state, and freeing the associated **mBlk** chains. The transmit descriptor clean routine uses the clean index to rotate through the driver's transmit descriptors. As the transmit descriptor clean routine moves around the ring, it determines if the descriptor currently referenced by the clean index has been released from the device transmit queue. If the indexed descriptor has been released from the device transmit queue, the transmit descriptor clean routine does whatever is necessary to put the descriptor back into a free state, and increments the free counter. The routine continues to traverse the ring until it encounters a descriptor that has not been released from the device transmit queue or until the free counter equals the number of transmit descriptors created by the device.

When the transmit descriptor clean routine determines that a descriptor has been released from the device transmit queue, it uses the clean index to reference the transmit packet association list. If the routine finds that the referenced transmit packet association list entry holds an **mBlk** pointer, it frees the **mBlk** chain with **netMblkClChainFree( )**.

### Implementing Checksum Offloading

Checksum offloading for legacy END drivers is handled in a manner similar to that of VxBus network interface drivers. For a complete discussion of checksum offloading, see *VxWorks Device Driver Developer's Guide (Vol. 2): Network Drivers*.

**NOTE:** Prior to VxWorks 6.5, support for checksum offloading is included in the network stack by default. In later releases, this feature must be enabled in the Wind River Network Stack. For more information, see the Wind River Network Stack documentation and your Platform release notes.

### Implementing Required Entry Points and Structures

This section describes the API for an END driver. It describes the structures that are essential to such a driver and the entry points you must implement in the driver.

**NOTE:** The organization of an END driver does not follow the model for a standard VxWorks I/O driver. The driver is not accessible through the **open( )** routine or other file I/O routines. The driver is organized to communicate with the MUX. The MUX then handles communication with the network protocols.

#### Required Structures for a Driver

Within your driver, you must allocate and initialize an **END_OBJ**. Your driver also needs to allocate and initialize the structures referenced in **END_OBJ** structures, such as **DEV_OBJ**, **NET_FUNCS**, and **M2_INTERFACETBL**. To pass packets up to the MUX, use an **mBlk** structure.

##### Providing Network Device Abstraction: END_OBJ

Your *end***Load( )** entry point must allocate, initialize, and return an **END_OBJ** structure. The MUX uses this **END_OBJ** structure as a place to store the tools it needs to manipulate the stack and the device driver. These tools include data as well as pointers to routines. The **END_OBJ** structure is defined in **end.h** as follows:

```
typedef struct end_object
    {
    NODE               node;        /* root of the device hierarchy */
    DEV_OBJ            devObject;   /* accesses your device's ctrl struct */
    FUNCPTR            receiveRtn;  /* routine to call on reception */
    BOOL               attached;    /* indicates unit is attached */
    SEM_ID             txSem;       /* transmitter semaphore */
```

```
long                flags;       /* various flags */
struct net_funcs    *pFuncTable; /* function table */
M2_INTERFACETBL     mib2Tbl;     /* MIBII counters */
struct ETHER_MULTI *pAddrList;   /* head of the multicast address list */
int                 nMulti;      /* number of elements in the list */
LIST                protocols;   /* protocol node list */
BOOL                snarfProto;  /* is someone snarfing us? */
void*               pMemPool;    /* memory cookie used by MUX bufr mgr. */
M2_ID*              pMib2Tbl;    /* RFC 2233 MIB objects */
} END_OBJ;
```

Your driver must set and manage some of these members. Other members are MUX-managed. To know which are which, read the following member descriptions:

**node**
The root of the device hierarchy. The MUX sets the value of this member. Your driver should treat it as opaque.

**devObject**
The **DEV_OBJ** structure for this device. Your driver must set this value at load time. See *Tracking Your Device's Control Structure: DEV_OBJ*, p.61.

**receiveRtn**
A function pointer that references a **muxReceive( )** routine. The MUX supplies this pointer by the completion of the **muxDevLoad( )** call that loads this driver. Your driver uses this function pointer to pass data up to the protocol.

**attached**
A **BOOL** indicating whether or not the device is attached. The MUX sets and manages this value.

**txSem**
A semaphore that controls access to this device's transmission facilities. The MUX sets and manages this value.

**flags**
A value constructed from **OR**ing in **IFF_*** flag constants. Except for **IFF_LOAN** and **IFF_SCAT**, these constants are the same **IFF_*** flags associated with the TCP/IP stack.

**IFF_UP**
The interface driver is up.

**IFF_BROADCAST**
The broadcast address is valid.

**IFF_DEBUG**
Debugging is on.

**IFF_LOOPBACK**
This is a loopback net.

**IFF_POINTOPOINT**
The interface is a point-to-point link.

**IFF_NOTRAILERS**
The device must avoid using trailers.

**IFF_RUNNING**
The device has successfully allocated needed resources.

**IFF_NOARP**
There is no address resolution protocol.

**IFF_PROMISC**
This device receives all packets.

**IFF_ALLMULTI**
This device receives all multicast packets.

**IFF_OACTIVE**
Transmission in progress.

**IFF_SIMPLEX**
The device cannot hear its own transmissions.

**IFF_LINK0, IFF_LINK1, IFF_LINK2**
Per link layer defined bits.

**IFF_MULTICAST**
The device supports multicast.

**IFF_LOAN**
The device supports buffer loaning.

**IFF_SCAT**
The device supports scatter-gather.

**pFuncTable**
A pointer to a **NET_FUNCS** structure. This structure contains function pointers to your driver's entry points for handling standard requests such as unload or send. Your driver must allocate and initialize this structure when the device is loaded. See *Identifying the Entry Points into Your Network Driver: NET_FUNCS*, p.63.

**mib2Tbl**
An **M2_INTERFACETBL** structure for tracking the MIB-II variables used in your driver. Your driver must initialize the structure referenced here, although both your driver and the MUX later adjusts the values stored in the table.

**NOTE:** The **mib2Tbl** field is retained for backwards compatibility with RFC 1213. Wind River does not recommended this field for new drivers. For new drivers, use the RFC 2233 interface.

**pAddrList**
A pointer to the head of a list of multicast addresses. The MUX sets and manages this list, but it uses your driver's *end***MCastAddrAdd( )**, *end***MCastAddrDel( )**, and *end***MCastAddrGet( )** entry points to do so.

**nMulti**
A value indicating the number of addresses on the list referenced in the **multiList** member. The MUX sets this value using the information returned by your driver's *end***MCastAddrGet( )**.

**protocols**
The head of the list of protocols that have bound themselves to this network driver. The MUX manages this list.

**snarfProto**
A **BOOL** indicating whether a packet-snarfing protocol has bound itself to this driver. Such a protocol can prevent the packet from passing on to lower priority protocols (see *Protocol Startup*, p.14). The MUX sets and manages this value.

**pMemPool**
A pointer to a **netBufLib**-managed memory pool. The MUX sets the value of this member. Your driver should treat it as opaque.

**pMib2Tbl**
The interface table for RFC 2233 compliance.

#### Tracking Your Device's Control Structure: DEV_OBJ

Your driver uses the **DEV_OBJ** structure to tell the MUX the name of your device and to hand the MUX a pointer to your device's control structure. This control structure is a device-specific structure that you define according to your needs. Your driver uses this control structure to track things such as flags, memory pool addresses, and so on. The information stored in the control structure is typically

essential to just about every driver entry point. The **DEV_OBJ** structure is defined in **end.h** as follows:

```
typedef struct dev_obj
    {
    char name[END_NAME_MAX];         /* device name */
    int unit;                        /* to support multiple units */
    char description[END_DESC_MAX];  /* text description */
    void* pDevice;                   /* pointer back to the device data. */
    } DEV_OBJ;
```

**name**

A pointer to a string of up to eight characters. This string specifies the name for this network device.

**pDevice**

A pointer to your driver's internal control structure. This field was originally intended as a back pointer to the driver control structure. The driver used this field to dereference itself from the **pCookie** passed from MUX calls. However, in a properly initialized END driver, this field is **NULL**. This is because an END driver should pass the **END_OBJ_INIT** macro **NULL** as the **pDevice** argument. The reason for this is that passing the device's control structure pointer results in the MUX freeing the structure when the device is unloaded from the MUX. Because the driver stores other ancillary pointers in its control structure (which it cannot free until after it has been unloaded from the MUX), it must preserve this pointer. The pointer is preserved by passing the **NULL** as **pDevice** in **END_OBJ_INIT**. Therefore, this field is deprecated and should not be used unless a driver allocates **END_OBJ** separately from its control structure (this practice is not recommended).

**unit**

This is the unit number for the particular named device. Unit numbers start at 0 and increase for every device controlled by the same driver. For example, if a system has two Lance Ethernet devices (named **ln**) then the first one is **ln0** and the second is **ln1**. If the same system also has a DEC 21x40 Ethernet, that device (whose name is **dc**) is **dc0**.

**description**

This is a text description of the device driver. For example, the **fei82557End** driver puts the string, "Intel 82557 Ethernet Enhanced Network Driver" into this location. This string is displayed if **muxShow( )** is called.

### Identifying the Entry Points into Your Network Driver: NET_FUNCS

The MUX uses the **NET_FUNCS** structure to maintain a table of entry points into your END driver. The **NET_FUNCS** structure is defined as follows:

```
typedef struct net_funcs
    {
    STATUS (*start) (void*);                /* driver's start func */
    STATUS (*stop) (void*);                 /* driver's stop func */
    STATUS (*unload) (void*);               /* driver's unload func */
    int (*ioctl) (void*, int, caddr_t);     /* driver's ioctl func */
    STATUS (*send) (void* , M_BLK_ID);      /* driver's send func */
    STATUS (*mCastAddrAdd) (void*, char*); /* driver's mcast add func */
    STATUS (*mCastAddrDel) (void*, char*); /* driver's mcast delete func */
    STATUS (*mCastAddrGet) (void*, MULTI_TABLE*);
                                            /* driver's mcast get func */
    STATUS (*pollSend) (void*, M_BLK_ID);   /* driver's poll send func */
    STATUS (*pollRcv) (void*, M_BLK_ID);    /* driver's poll receive func */
    STATUS (*addressForm) (M_BLK_ID, M_BLK_ID, M_BLK_ID);
                                            /* driver's addr formation func */
    STATUS (*packetDataGet) (M_BLK_ID, M_BLK_ID);
                                            /* driver's pkt data get func */
    STATUS (*addrGet) (M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
                                            /* driver's pkt addr get func */
    } NET_FUNCS;
```

Within your *end***Load( )** routine, initialize these members to point to the appropriate driver entry points. Thus, **start** should contain a pointer to your *end***Start( )**, **stop** to your *end***Stop( )**, **unload** to your *end***Unload( )**, and so on.

### Tracking Link-Level Information: LL_HDR_INFO

The MUX uses **LL_HDR_INFO** structures to keep track of link-level header information associated with packets passed from an END driver to the MUX and from there up to a protocol. An **LL_HDR_INFO** structure is passed as an argument to all stack receive routines (see, *Passing a Packet Up to the Protocol: stackRcvRtn( )*, p.19).

```
typedef struct llHdrInfo
    {
    int destAddrOffset;  /* destination addr offset in mBlk */
    int destSize;        /* destination address size */
    int srcAddrOffset;   /* source address offset in mBlk */
    int srcSize;         /* source address size */
    int ctrlAddrOffset;  /* control info offset in mBlk */
    int ctrlSize;        /* control info size */
    int pktType;         /* type of the packet */
    int dataOffset;      /* data offset in the mBlk */
    } LL_HDR_INFO;
```

**destAddrOffset**
Offset into **mBlk** structure at which the destination address starts.

**destSize**
Size of destination address.

**srcAddrOffset**
Offset into **mBlk** structure at which the source address starts.

**srcSize**
Size of source address.

**ctrlAddrOffset**
Reserved for future use.

**ctrlSize**
Reserved for future use.

**pktType**
Type of packet. For a list of valid packet types, see RFC 1700.

**dataOffset**
Offset into **mBlk** structure at which the packet data starts.

#### Tracking Data That Passes Between the Driver and the Protocol: mBlk

Use **mBlk** structures as a vehicle for passing packets between the driver and protocol layers. The **mBlk** structure is defined in **netBufLib.h** as follows:

```
typedef struct mBlk
    {
    M_BLK_HDR   mBlkHdr;                /* header */
    M_PKT_HDR   mBlkPktHdr;             /* pkthdr */
    CL_BLK *    pClBlk;                 /* pointer to cluster blk */
    } M_BLK;
```

**mBlkHdr**
Contains a pointer to an **mHdr** structure. For the most part, you should have no need to access or set this member directly and can treat it as opaque. The only exception is when you must chain this **mBlk** to another. In that case, you need to set the value of **mBlk.mHdr.mNext** or **mBlk.mBlkHdr.mNextPkt** or both. Use **mBlk.mBlkHdr.mNext** to point to the next **mBlk** in a chain of **mBlk**s. Use **mBlk.mHdr.mNextPkt** to point to an **mBlk** that contains the head of the next packet.

**mBlkPktHdr**
Contains a pointer to a **pktHdr** structure. You should have no need to access or set this member directly and can treat it as opaque.

**pClBlk**
Contains a pointer to a **clBlk** structure. You should have no need to access or set this member directly and can treat it as opaque. However, if you are not

using **netBufLib** to manage the driver's memory pool, you must provide your own memory free routine for its associated cluster. To do this, you must update **mBlk.pClBlk.pClFreeRtn** to point to your customized free routine. This routine must use the same API as the **netBufLib** free routine. This means that the **mBlk.pClBlk.pFreeArg1**, **mBlk.pClBlk.pFreeArg2**, and **mBlk.pClBlk.pFreeArg3** members must also be updated.

Setting appropriate values for the members listed above (and the members of all the referenced structures) is just a matter of calling the appropriate **netBufLib** routines for the creation of an **mBlk**/**clBlk**/cluster construct (or tuple). For more information, see *Setting Up and Using Memory for Receive and Transmit Buffers*, p.34.

### Required Driver Entry Points

The names of all entry points described in this section begin with the prefix *end*. This indicates that they are generic driver entry points. Within your particular network driver, the specific entry points should use a prefix that indicates the driver of which they are a part. For example, you would use an **ln** prefix in the entry points associated with the AMD Lance driver. Thus, your network interface driver would define the entry points **lnLoad( )**, **lnUnload( )**, **lnReceive( )**, and so on.

This naming convention for driver entry points is a matter of good coding practice. Because VxWorks references these entry points using the function pointers you load into a **NET_FUNCS** structure, you are free to follow other conventions for assigning names to entry points.

Table 3-2 **Required Driver Entry Points**

| Routine | Purpose |
|---|---|
| *end***Load( )** | Initialize the driver and load it into the MUX. |
| *end***Unload( )** | Free driver resources. |
| *end***Start( )** | Start the driver. |
| *end***Stop( )** | Stop the driver. |
| *end***Send( )** | Send a packet out on the hardware. |
| *end***Ioctl( )** | Access driver control routines. |
| *end***MCastAddrAdd( )** | Add an address to the device's multicast address list. |

Table 3-2 **Required Driver Entry Points** (cont'd)

| Routine | Purpose |
|---|---|
| *end***MCastAddrDel( )** | Delete an address from the device's multicast address list. |
| *end***MCastAddrGet( )** | Get the list of multicast addresses maintained for this device. |
| *end***PollSend( )** | Do a polling send. |
| *end***PollReceive( )** | Do a polling receive. |
| *end***AddressForm( )** | Add the appropriate link-level information into an **mBlk** in preparation for transmission. This routine is provided by the network stack and not typically defined by the driver. |
| *end***PacketDataGet( )** | Extract packet data (omitting link-level information) from one **mBlk** and write it to another. This routine is provided by the network stack and not typically defined by the driver. |
| *end***PacketAddrGet( )** | Extract address information (omitting packet data) from one **mBlk** and write out each source and destination address to its own **mBlk**. For an Ethernet packet, this requires two output **mBlk**s. However, for some non-Ethernet packets, this could require as many as four output **mBlk**s because the local source and destination addresses can differ from the ultimate source and destination addresses. This routine is provided by the network stack and not typically defined by the driver. |

## External Interface

### Loading the Device: endLoad( )

The routine *end***Load( )** handles parameter parsing, configuration, and initialization. *end***Load( )** is the initial entry point into every network interface driver. The **tUserRoot** task specifies your *end***Load( )** as an input parameter when it calls **muxDevLoad( )** to load your driver.

Your *end***Load( )** routine must take the following form:

```
END_OBJ* endLoad
    (
    char* initString /* a string encoded for the device to use for its */
                     /* initialization arguments. */
    )
```

Within the *end***Load( )** routine, you must handle any device-specific initialization. You should also set values for most of the members of the **END_OBJ** structure. Of particular interest are the **END_OBJ** members **receiveRtn**, **pFuncTable**, and **devObject**. For more information on these members, see the member descriptions provided in *Providing Network Device Abstraction: END_OBJ*, p.58.

*end***Load( )** should return a pointer to an initialized **END_OBJ** structure. If an error occurs, return **ERROR**.

The argument is:

**initString**
Passes in any initialization parameters needed.

The *end***Load( ) initString** argument is a pointer to a tokenized string of driver configuration parameters. Each parameter is delineated by a colon (:). The *end***Load( )** routine parses the **initString** argument and stores it in its driver control structure. The routine first allocates memory for the driver control structure and then passes a pointer to the driver control structure along with the pointer to **initString**, to a parser that breaks the parameters down into discrete values and loads them into the driver control structure.

During system initialization, the operating system calls this routine two times for every matching interface configured into the system. In the first call, the OS passes a pointer to a null string to the driver, and the driver is responsible for filling the string with the device name. The second call is when actual device and driver initialization takes place.

Near the beginning of the *end***Load( )** routine, there is usually code similar to the following:

```
END_OBJ * templateEndLoad
    (
    char *initString     /* parameter string */
    )
    {
    DRV_CTRL * pDrvCtrl; /* pointer to DRV_CTRL structure */
    ...
    if (initString == NULL)
        return (NULL);
```

```
    if (initString[0] == 0)
        {
        bcopy ((char *)DEV_NAME, (void *)initString, DEV_NAME_LEN);
        return (0);
        }
    }
```

*end***Load( )** configures the device's registers to either the default values or as prescribed by the driver parameters.

*end***Load( )** calls a memory initialization routine that allocates a contiguous amount of memory for DMA descriptors, the amount allocated is determined by the number of descriptors specified in the parameters, or a default value defined in the driver. The memory initialization routine also calls **netPoolCreate( )** in **netBufLib**, this routine creates a tuple pool sufficient for the driver's needs.

The memory initialization routine initializes the driver's DMA descriptors. It organizes the descriptors as indicated by the device's specification. The routine accesses each discrete descriptor and fills the descriptor fields according to the device's expectations and the driver's parameter instructions. In the case of receive descriptors, it also obtains a tuple from the **netPool** it created, writes the tuple's cluster buffer pointer into the descriptor, and stores the tuple's **mBlk** pointer in the driver's association list. This is a convenient location from which it can later be correlated back to the descriptor's DMA buffer.

Additional routines are necessary for network stack operations. Entry points to these routines are provided by the **NET_FUNCS** structure, which is pointed to by an entry in the **END_OBJ** structure. Normally, these routines are declared local to the driver and are only accessed through the **NET_FUNCS** structure. For a description of the driver routines, see Table 3-2.

#### Unloading the Device: endUnload( )

Your *end***Unload( )** entry point should handle everything needed to remove this network driver from the system. Within your *end***Unload( )** routine, you should handle things such as cleanup for all of the local data structures. Your *end***Unload( )** routine does not need to worry about notifying protocols about unloading the device. Before calling *end***Unload( )**, the MUX sends a shutdown notice to each protocol attached to the device. However, you must be sure to delete any semaphores that are created in the driver.

*end***Unload( )** must take the following form:

```
void endUnload
    (
    void* pCookie /* pointer to device-identifying END_OBJ */
    )
```

This routine is declared as **void** and thus should return no function value.

The parameters are:

**pCookie**
Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. You should probably free the associated memory from this routine in your *end***Unload( )** routine.

- **Unloading an END Driver**

**NOTE:** This example assumes a VxWorks 6.*x* environment and the use of **netPoolCreate( )** to establish the driver's buffer pool. Also, to use these instructions, an END driver must pass a **NULL** as the second argument to **END_OBJ_INIT**.

The unload routine in an END driver can only be called through **muxUnload( )**. Before the **muxUnload( )** routine calls the driver's unload routine, it must unbind the device driver from any protocols to which it was previously bound. The driver's unload routine must then complete the unload by:

- disabling the device
- freeing its associated memory

The unload must complete these steps in an order that prevents a memory access to already freed memory as well as prevents the loss of any pointers. This means:

- the DMA engine must be stopped and interrupts disabled before the receive ring is dismantled
- the driver must be unbound from the MUX before the transmit queue and its semaphore are dismantled or freed
- all memory loaned from the driver's pool must be returned before it is freed
- because the driver's control structure stores all the pointers for these regions, it must be the last resource to be freed

All END drivers cause four instances of memory allocation. These instances are as follows:

- the driver control structure stored in **pDrvCtrl**
- the transmit semaphore stored in **pDrvCtrl->endObj.txSem**
- the transmit and receive descriptors
- tuples (clusters, **mBlk**s, and **clBlk**s)

**NOTE:** It is also possible that some END drivers employ one or more watchdog timers. These timers must also be deleted.

**NOTE:** If an END driver allocates any additional memory, it is the responsibility of the END driver to free that memory when it is unloaded.

Each of these instances of memory can only be freed after:

- there is no possibility of the memory being inadvertently accessed
- the memory is not holding the only copy of a pointer to allocated memory

These conditions impose a specific sequence of events for freeing the memory areas:

1. To ensure that there are no more interrupts generated by the device, stop the device's DMA engine and disable all of the device interrupts.
2. Call **wdDelete( )** for any watchdog timers associated with the driver.
3. Ensure that all transmit descriptors are cleaned and the associated tuples are freed.
4. Free the transmit semaphore.
5. Ensure that the driver has relinquished *all* tuples and individual clusters, **mBlk**s, and **clBlk**s back to the pool. That is, ensure that:
   - all receive descriptors have had their associated tuples freed back to the driver's pool
   - any buffers or tuples used for polling mode are also freed back to the driver's pool
6. Free transmit and receive descriptors.
7. Call the **netPoolRelease( )** routine to ensure that the **netBufLib** frees the driver's pool memory back to the heap when all clusters, **mBlk**s, and **clBlk**s are returned to the pool.
8. Free the driver's control structure.

   **NOTE:** The macro call to **END_OBJ_INIT** must have a **NULL** as its second argument. Otherwise, the MUX attempts to free the driver's control structure resulting in a double free error.

9. Exit the unload routine.

### Providing an Opaque Control Interface to Your Driver: endIoctl( )

Your *end***Ioctl( )** entry point should handle all requests for changes to the state of the device, such as bringing it up, shutting it down, turning on promiscuous mode, and so on. You can also use your *end***Ioctl( )** routine to provide access to MIB-II interface statistics.

Your *end***Ioctl( )** must take the following form:

```
STATUS endIoctl
    (
    void* pCookie, /* pointer to device-identifying END_OBJ */
    int cmd,       /* value identifying command */
    caddr_t data   /* data needed to complete command */
    )
```

If there are no errors, this routine should return **OK**. If errors occur, one of the following values should be returned:

**EINVAL**
The **ioctl( )** command is not supported or an argument is not valid.

**ENOTSUP**
The device is not capable of supporting the requested command, or has been configured not to support the requested command. This happens with the **EIOCGMEDIALIST**, for example, when the media list is empty.

**ENOSPC**
The driver cannot perform the requested command due to lack of an available buffer, lack of space in a ring buffer, full list, or other lack of a required resource.

For some commands, this routine may return the return value of some utility routine, such as the return value from *end***M2Ioctl( )** for the **EIOCGMIB2233** and **EIOCGMIB2** commands.

The parameters are:

**pCookie**
Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**.

**cmd**
Can pass any of the values shown in the command column of Table 3-3. Your *end***Ioctl( )** must have an appropriate response to each command.

**data**
Passes the data, or a pointer to the data, that your *end***Ioctl( )** needs to carry out the command specified in **cmd**.

Table 3-3 **ioctl( ) Commands and Data Types**

| Command | Function | Data Type |
|---|---|---|
| **EIOCSFLAGS** | Set device flags. | **int**; see description of **END_OBJ.flags** |
| **EIOCGFLAGS** | Get device flags. | **int** |
| **EIOCSADDR** | Set device address. | **char*** |
| **EIOCGADDR** | Get device address. | **char*** |
| **EIOCMULTIADD** | Add multicast address. | **char*** |
| **EIOCMULTIDEL** | Delete multicast address. | **char*** |
| **EIOCMULTIGET** | Get multicast list. | **MULTI_TABLE*** |
| **EIOCPOLLSTART** | Set device into polling mode. | **NULL** |
| **EIOCPOLLSTOP** | Set device into interrupt mode. | **NULL** |
| **EIOCGPOLLCONF** | Configure a data location from which the network stack can read statistics | **END_IFDRVCONF*** |
| **EIOCGPOLLSTATS** | Return network statistics to the caller | **END_IFCOUNTERS*** |
| **EIOCGFBUF** | Get minimum first buffer for chaining. | **int** |
| **EIOCGMIB2** | Get the MIB-II counters from the driver. | **M2_INTERFACETBL*** |

#### Sending Data Out on the Device: endSend( )

The MUX calls your *end***Send( )** entry point when it has data to send out on the device. Your *end***Send( )** routine must take the following form:

```
STATUS endSend
    (
    void* pCookie,          /* device structure */
    M_BLK_ID pMblk,         /* data to send */
    )
```

This routine should return **OK**, **ERROR**, or **END_ERR_BLOCK**.

The value **END_ERROR_BLOCK** should be returned if the packet cannot be transmitted at this time because it is in polling mode, or because of a lack of resources. In either case, the packet is not freed from the **mBlk** chain.

The value **OK** is returned upon successful acceptance of the data packet. If an error occurs, **ERROR** is returned and **errno** should be set. In these cases, the data packet is freed from the **mBlk** chain.

The parameters are:

**pCookie**
Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pMblk**
Passes a pointer to an **mBlk** structure containing the data you want to send. For more information on how to setup an **mBlk**, see *Setting Up and Using Memory for Receive and Transmit Buffers*, p.34.

In most cases, a transmit-done interrupt routine schedules a task-level routine to free the **mBlk** after the packet is sent.

#### Starting a Stopped but Loaded Driver: endStart( )

Your *end***Start( )** entry point should do whatever is necessary to make the driver active. For example, it should register your device driver's interrupt service routine.Your *end***Start( )** routine must take the following form:

```
Status endStart
    (
    void* pCookie /* pointer to device-identifying END_OBJ structure */
    )
```

This routine should return **OK** or **ERROR**. If an error occurs, the routine should set **errno**.

The parameters are:

**pCookie**
Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

However, your *end***Start( )** should probably include this pointer as a parameter to the **sysIntConnect( )** routine that it uses to register the ISR. The ISR may not

have any direct use for the **END_OBJ** pointer, but it should pass the pointer into the driver entry point that handles task-level processing for packet reception.

When it comes time to pass the packet up to the MUX, your driver must call the MUX-supplied routine referenced in **pCookie.receiveRtn**. See *Providing Network Device Abstraction: END_OBJ*, p.58.

#### Stopping the Driver Without Unloading It: endStop( )

Your *end***Stop( )** entry point can assume that the driver is already loaded and that *end***Load( )** has already been called. Within your *end***Stop( )** routine, you should do whatever is necessary to make the driver inactive without actually unloading the driver. *end***Stop( )** must take the following form:

```
STATUS endStop
    (
    void* pCookie  /* pointer to a device-identifying END_OBJ structure */
    )
```

This routine should return **OK** or **ERROR**. If an error occurs, the routine should set **errno**.

The parameters are:

**pCookie**
Passes in a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

#### Handling a Polled Send: endPollSend( )

The *end***PollSend( )** routine is intended for use by the debug agent during *system mode*—that is, when the kernel is stopped. Because the kernel is unavailable in system mode, this entry point cannot make any system calls. Likewise, this entry point should not block because it could result in a system failure or *hang*.

*end***PollSend( )** must take the following form:

```
STATUS endPollSend
    (
    void* pCookie,      /* pointer to device-identifying END_OBJ structure */
    M_BLK_ID pMblk,     /* data to send */
    )
```

Within your *end***PollSend( )** routine, check that the device is set to polled mode (by a previous *end***Ioctl( )** call). Wind River recommends that your *end***PollSend( )** routine keep a transmit tuple, allocated from the driver's pool, permanently

available for its use. The pointer to this tuple should be stored in driver's **DRV_CTRL** structure.

```
if ((pDrvCtrl->pTxPollMblk = netTupleGet (pDrvCtrl->endObj.pNetPool,
                                ETHERMTU + /* max data portion */
                                16       + /* size of enet header */
                                4,         /* FCS */
                                M_DONTWAIT, MT_DATA, FALSE)) == NULL)

    {
    pDrvCtrl->lastError.errCode = END_ERR_NO_BUF;
    muxError(&pDrvCtrl->endObj, &pDrvCtrl->lastError);

    return ERROR;
    }
```

Then, keep a pointer to the transmit tuple's cluster buffer as follows:

```
pDrvCtrl->pTxPollBuf = (UCHAR *)pDrvCtrl->pTxPollMblk->mBlkHdr.mData;
```

The *end***PollSend( )** routine should use the **netMblkToBufCopy( )** utility to copy **pMblk** to its polling buffer. The *end***PollSend( )** routine should then put the **pTxPollMblk** onto the next available descriptor on the device's output queue.

```
len = netMblkToBufCopy (pMblk, (char *) pDrvCtrl->pTxPollBuf, NULL);
```

The *end***PollSend( )** routine and the *end***Send( )** routine share the same transmit descriptors and the same transmit queue. Therefore, *end***PollSend( )** should treat the transmit queue and descriptors in the same manner as the *end***Send( )** routine.

This routine should return **OK** or **ERROR**. If an error occurs, the routine should set **errno**.

The parameters are:

**pCookie**
: Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pMblk**
: Passes a pointer to an **mBlk** structure containing the data you want to send. For information on setting up an **mBlk**, see *Setting Up and Using Memory for Receive and Transmit Buffers*, p.34.

#### Handling a Polled Receive: endPollReceive( )

The *end***PollReceive( )** routine is intended for use by the debug agent during *system mode*—that is, when the kernel is stopped. Because the kernel is unavailable in

system mode, this entry point cannot make any system calls. Likewise, this entry point should not block because it could result in a system failure or *hang*.

*end***PollReceive( )** must take the following form:

```
int endPollReceive
    (
    void* pCookie,        /* device structure */
    M_BLK_ID pMblk        /* place to return the data */
    )
```

Your *end***PollReceive( )** routine should check that the device is set to polled mode (by a previous *end***Ioctl( )** call). Your *end***PollReceive( )** should then get a packet directly from the network and copy it to the **mBlk** passed in by the **pMblk** parameter.

Your *end***PollReceive( )** entry point should return **OK** or an appropriate error value. One likely error return value is **EAGAIN**. Your routine should return **EAGAIN** if the submitted **mBlk** is not big enough to contain the received packet, or if no packet is available.

The parameters are:

**pCookie**
: Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pMblk**
: Passes in a pointer to an **mBlk** structure. This parameter is an output parameter. Your *end***PollReceive( )** routine must copy the data from the stack to the **mBlk** structure referenced here.

#### Adding a Multicast Address: endMCastAddrAdd( )

Your *end***MCastAddAddr( )** entry point must add an address to the multicast table that is maintained by the device. *end***MCastAddAddr( )** must take the following form:

```
STATUS endMCastAddAddr
    (
    void* pCookie,   /* pointer to a device-identifying END_OBJ structure */
    char* pAddress   /* pointer to address to add */
    )
```

To help you manage a list of multicast addresses, VxWorks provides the library **etherMultiLib**.

This routine should return **OK** or **ERROR**. If an error occurs, the routine should set **errno**.

The parameters are:

**pCookie**
Passes in a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pAddress**
Passes in a pointer to the address you want to add to the list. To help you manage a list of multicast addresses, VxWorks includes the library, **etherMultiLib**.

Within your *end***MCastAddrAdd( )** routine, you must reconfigure the interface in a hardware-specific way. This reconfiguration should allow the driver to receive frames from the specified address and then pass those frames up to the higher layer.

#### Deleting a Multicast Address: endMCastAddrDel( )

Your *end***MCastAddrDel( )** entry point must delete an address from the multicast table maintained by the device. *end***MCastAddrDel( )** must take the following form:

```
STATUS endMCastAddrDel
    (
    void* pCookie,    /* pointer to a device-identifying END_OBJ structure */
    char* pAddress    /* pointer to address to delete */
    )
```

This routine should return **OK** or **ERROR**. If an error occurred, the routine should set **errno**.

The parameters are:

**pCookie**
Passes a pointer to the **END_OBJ** structure returned by *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pAddress**
Passes a pointer to the address you must delete. To help you manage a list of multicast addresses, VxWorks includes the library, **etherMultiLib**.

Your *end***MCastAddrDel( )** must also reconfigure the driver (in a hardware-specific way) so that the driver no longer receives frames with the specified address.

#### Getting the Multicast Address Table: endMCastAddrGet( )

Your *end***MCastAddrGet( )** routine must get a table of multicast addresses and return it in the buffer referenced in the **pMultiTable** parameter. These addresses are the list of multicast addresses which the interface is currently monitoring. Your *end***MCastAddrGet( )** must take the following form:

```
STATUS endMCastAddrGet
    (
    void* pCookie,
    MULTI_TABLE* pMultiTable
    )
```

To get the list of multicast address, use the routines provided in **etherMultiLib**.

This routine should return **OK** or **ERROR**. If an error occurs, the routine should set **errno**.

The parameters are:

**pCookie**
Passes in a pointer to the **END_OBJ** structure you returned from *end***Load( )**. Because the first field in the driver's control structure (**DRV_CTRL**) is always **END_OBJ**, most drivers expect **pDrvCtrl**. This is allowed because **pCookie** and **pDrvCtrl** are interchangeable.

**pMultiTable**
Passes in a pointer to a buffer. This is an output parameter. Your *end***MCastAddrGet( )** routine must write a **MULTI_TABLE** structure into the referenced buffer. **end.h** defines **MULTI_TABLE** as follows:

```
typedef struct
    {
    long len;         /* length of table in bytes */
    char *pTable;     /* pointer to entries */
    } MULTI_TABLE;
```

Modify the **len** member of the **MULTI_TABLE** to indicate just how many addresses you are returning. Write the addresses to the buffer referenced in the **pTable** member of the **MULTI_TABLE**.

#### Forming an Address for Packet Transmission: endAddressForm( )

The *end***AddressForm( )** routine must take a source address and a destination address and copy the information into the data portion of the **mBlk** structure in a

fashion appropriate to the link level. Implementing this functionality is the responsibility of the driver writer. However, a simple implementation of this routine is provided in **endLib**, you can use this routine as provided and are not required to provide your own. After adding the addresses to **mBlk**, your *end***AddressForm( )** routine should adjust the **mBlk.mBlkHdr.mLen** and **mBlk.mBlkHdr.mData** members accordingly. This routine must take the following form:

```
M_BLK_ID endAddressForm
    (
    M_BLK_ID pMblk,        /* packet data */
    M_BLK_ID pSrcAddress,  /* source address */
    M_BLK_ID pDstAddress   /* destination address */
    )
```

This routine returns an **M_BLK_ID**, which is potentially the head of a chain of **mBlk** structures.

If the cluster referenced by **pMblk** does not have enough room to contain both the header and the packet data, this routine must reserve an additional tuple (**mBlk**/**clBlk**/cluster construct) to contain the header. This routine must then chain the **mBlk** in **pMblk** onto the just-reserved header **mBlk** and return a pointer to the header **mBlk** as the function value.

The parameters are:

**pMblk**
The **mBlk** that contains the packet to be transmitted.

**pSrcAddress**
The **mBlk** that contains the link-level address of the source.

**pDstAddress**
The **mBlk** that contains the link-level address of the destination.

#### Getting a Data-Only mBlk: endPacketDataGet( )

The *end***PacketDataGet( )** routine must provide a duplicate **mBlk** that contains the packet data in the original but skips the header information. Some common cases are provided for in **endLib**. This routine should return **OK** or **ERROR** and set **errno** if an error occurs.

The routine is of the following form:

```
STATUS endPacketDataGet
    (
    M_BLK_ID pBuff,            /* packet data and address information */
    LL_HDR_INFO* pLinkHdrInfo  /* structure to hold link-level info. */
    )
```

The parameters are:

**pBuff**
Expects a pointer to the **mBlk** that contains both header and packet data.

**pLinkHdrInfo**
Returns an **LL_HDR_INFO** structure containing header information that is dependent upon the particular data-link layer that the END driver implements. For more information, see *Tracking Link-Level Information: LL_HDR_INFO*, p.63.

#### Return Addressing Information: endEtherPacketAddrGet( )

The *end***EtherPacketAddrGet( )** routine locates the addresses in a packet. This routine takes an **M_BLK_ID**, locates the address information, and adjusts the **M_BLK_ID** structures referenced in **pSrc**, **pDst**, **pESrc**, and **pEDst** so that their **pData** members point to the addressing information in the packet. The addressing information is not copied. All **mBlk** structures share the same cluster.

```
STATUS endEtherPacketAddrGet
    (
    M_BLK_ID pMblk, /* pointer to packet */
    M_BLK_ID pSrc,  /* pointer to source address */
    M_BLK_ID pDst,  /* pointer to destination address */
    M_BLK_ID pESrc, /* pointer to source address (if any) */
    M_BLK_ID pEDst  /* pointer to destination address (if any) */
    )
```

**pSrc**
Expects **NULL** or a pointer to the **mBlk** structure into which to write the extracted source address of the packet.

**pDst**
Expects **NULL** or a pointer to the **mBlk** structure into which to write the extracted destination address of the packet.

**pESrc**
Expects **NULL** or a pointer to the **mBlk** structure into which to write the extracted source of the packet.

**pEDst**
Expects **NULL** or a pointer to the **mBlk** structure into which to write the extracted destination address of the packet.

# 3.3 The END Driver Development Process



This section provides an overview of the END driver development process. At a high level, it provides the steps you should take when developing an END driver for use with VxWorks.

## 3.3.1 Driver Development Overview

This section provides a high-level overview of the steps required to write or port an END driver for VxWorks.

#### Writing a New Driver

**NOTE:** Wind River does not recommend using the legacy driver model for new development. For more information, see *1. Introduction*.

The first step in creating a new driver is to define the structure associated with each interface of the device. This structure must begin with an **END_OBJ** structure. This allows the driver to share its **END_OBJ** structure with the network stack by using a single pointer which points to both objects.

This structure should also contain a pointer to each register that the device contains, along with flags, data pointers, and other information specific to the interface. This structure may need to be modified during driver development to add fields for unforeseen requirements. For example:

```
typedef struct drv_ctrl
    {
    END_OBJ       endObj;         /* base class */
    int           unit;           /* unit number */
    ...                           /* other per-interface variables */
    } DRV_CTRL;
```

When writing a new driver, you should first focus on initialization code. Where appropriate, the low level device manipulation routines discussed in earlier sections can be used during initialization. Stubs for routines in the **NET_FUNCS** structure should be created, and the **NET_FUNCS** structure itself should be filled. The initialization code should disable interrupts and set the device to a quiescent state. That is, it must place the hardware in a state where it does not generate interrupts that the processor is unable to handle at this point in the system initialization process.

Buffer allocation is done during initialization. It is strongly recommended that **netPoolCreate( )** be used as described in *Setting Up a Memory Pool*, p.36. Buffer allocation creates clusters, **clBlk**s, and **mBlk**s for transferring packets between the driver and the network stack. Both **clBlk**s and **mBlk**s are used by the driver and the network stack, but they are not handled by the device. Clusters are used by the network stack, the driver, and the device. For this reason, caching is an important concern.

At the time the buffers are allocated, you should also decide what structures will be used by the device. The device can usually be configured to manipulate a list or ring of buffers. If possible, a ring is preferred. In addition, the code to manipulate clusters, **clBlk**s, and **mBlk**s should be tested at this time. You should take a great deal of care when creating the buffer manipulation code, as well as when designing the device structures.

If you are not working with an existing driver, you must now create the low level device manipulation code. If you are porting an existing driver, this step should already be done. In many cases, the low level device manipulation functionality should be implemented as macros.

The low-level code should include code to configure the device by reading and writing device registers. This includes items such as enabling and disabling interrupts, starting the device, resetting the device, disabling the device, setting addresses, and so forth. The low level code should also include code to manipulate the send and receive rings. Remember to use the routines **sysInByte( )**, **sysInWord( )**, **sysInLong( )**, **sysOutByte( )**, **sysOutWord( )**, and **sysOutLong( )** to manipulate the device registers. These should be set to macros in the header file so that the actual routines can be easily overridden when necessary. For example:

```
#ifndef TEMPLATE_BYTE_RD
#define TEMPLATE_BYTE_RD(addr, value) (value = sysInByte ((ULONG) addr))
#endif
```

Additional low level code is used to manipulate the device structures. For more information on structures, see *Implementing Required Entry Points and Structures*, p.58.

Next, write the polled mode input and output routines. This does not allow normal network traffic, but it can be used for system mode debugging as well as to test the functionality of the code used to manipulate the device. Remember that the polled receive routine must return immediately, whether a packet is available or not.

The interrupt code is developed after testing the polled mode routines. At this point, you know that you can manipulate the device correctly to send and receive packets, put buffers in the transmit ring, remove buffers from the receive ring, as well as start and stop the device.

### Porting an Existing Driver From Another OS

In general, device drivers provide code for manipulation of a device, and provide the interface between the driver and the OS. If you have a working, well-written driver from another OS, the device manipulation routines should be relatively easy to port.

It is vital to test the device on the original OS before beginning the porting effort. This insures that the driver is working correctly. Often, there are problems with the driver on the original OS. If these problems can be isolated before the porting effort, time is not wasted trying to debug the OS for an existing problem in the driver. If problems are found, you must decide to correct any problems on the original OS before the porting effort begins or begin the porting effort with the knowledge that you have a flawed driver. Correcting problems before the port makes the porting effort easier, but may delay partial availability of the driver on VxWorks. In either case, creating a list of existing problems should be considered a requirement before the porting effort begins.

In the best case, the low-level device manipulation routines can simply be copied from the existing driver into the new one. If the low-level device manipulation routines are not small, portable functions, it is probably worthwhile to extract the different areas of device-related functionality from the existing driver and create small modules for specific purposes. In many cases, the low-level device manipulation functionality should be implemented as macros. It may also be relatively straightforward to port the routines which manipulate the device structures.

Because of the unique interface between the driver and VxWorks, the remainder of the END driver port may be similar to writing a driver from scratch. Specifically, the initialization code, the receive routine, and the interrupt handlers require modification.

### Additional Development Issues

This sections highlights some additional development concerns that you may wish to consider before starting your driver development.

#### Backwards Compatibility

When writing a new driver for an initial revision of hardware, you can assume that this is not the only write of the driver. For this reason, care should be taken to accommodate future driver revisions. Often, a driver is upgraded to support a new revision of the hardware. In this case, care should be taken to ensure that the driver

is backwards compatible to both the older revisions of the driver and to existing BSPs that already use the driver.

#### Performance

A driver should minimize the use of **intLock( )**. The **intLock( )** routine has a negative performance impact on the entire system, and the impact can be significant. Normally, interrupts for the device are masked or interrupts for the given device are disabled. This is sufficient for most critical sections of code in a driver. By calling **intLock( )**, you are locking all interrupts and not just the Ethernet device interrupts.

Another performance concern is buffer copying. Buffer copying seriously impairs the performance of your driver and is typically unnecessary.

#### Common Problems

As with most driver development, care must be taken to ensure that structures are protected against corruption caused by concurrent access. This includes access from multiple VxWorks tasks as well as asynchronous access by the device.

### 3.3.2 Error Conditions

Sometimes an END driver encounters errors or other events that are of interest to the protocols using that END driver. For example, the device could go down, or the device can go down and then come back online. When such situations arise, the END driver should call **muxError( )**. This routine passes error information up to the MUX, which in turn passes the information on to all protocols that have registered a routine to receive the information. The **muxError( )** routine is declared as follows:

```
void muxError
    (
    void* pCookie,          /* pointer to END_OBJ */
    END_ERR* pError         /* pointer to END_ERR structure */
    )
```

Among its input, this routine expects a pointer to an **end_err** structure, which is declared in **end.h** as follows:

```
typedef struct end_err
    {
    INT32 errCode;           /* error code, see above */
    char* pMesg;             /* NULL-terminated error message, can be NULL */
    void* pSpare;            /* pointer to user defined data, can be NULL */
    } END_ERR;
```

The error-receive routine that the protocol registers with the MUX must be of the following prototype:

```
void xxError
    (
    END_OBJ* pEnd,   /* pointer to END_OBJ */
    END_ERR* pError, /* pointer to END_ERR */
    void* pSpare     /* pointer to protocol private data passed in muxBind */
    )
```

The **errCode** member of an **end_err** structure is 32 bits long. Wind River reserves the lower 16 bits of **errCode** for its own error messages. However, the upper 16 bits are available to user applications. Use these bits to encode whatever error messages you need to pass between drivers and protocols. The currently defined error codes are as follows:

```
#define END_ERR_INFO     1   /* information only */
#define END_ERR_WARN     2   /* warning */
#define END_ERR_RESET    3   /* device has reset */
#define END_ERR_DOWN     4   /* device has gone down */
#define END_ERR_UP       5   /* device has come back on line */
#define END_ERR_FLAGS    6   /* device flags have changed */
#define END_ERR_NO_BUF   7   /* device's cluster pool is exhausted */
```

These error codes have the following meaning:

**END_ERR_INFO**
This error is information only.

**END_ERR_WARN**
A non-fatal error has occurred.

**END_ERR_RESET**
An error occurred that forced the device to reset itself, but the device has recovered.

**END_ERR_DOWN**
A fatal error occurred that forced the device to go down. The device can no longer send or receive packets.

**END_ERR_UP**
The device was down but is now up again and can receive and send packets.

**END_ERR_BLOCK**
The device is busy, the transaction should be tried again later.

**END_ERR_FLAGS**
The device flags have changed.

**END_ERR_NO_BUF**
The device's cluster pool is exhausted.

### 3.3.3 Generic MIB Interface Initialization

The generic MIB interface used with VxWorks 6.*x* is an abstraction layer that supports either RFC 1213 or RFC 2233. This flexibility is required because the preprocessor cannot absolutely determine which type of MIB is in use. This uncertainty exists because components of the RFC 2233 MIB can be removed through the project facility and, because END drivers are precompiled and statically linked to the VxWorks image, they cannot use RFC 2233 MIB components which cannot be guaranteed to be present. This is problematic because the two interfaces employ different APIs. Therefore, because the drivers cannot reliably predict which API is present, the API must be abstracted.

The instructions in this section are intended for initially implementing the generic MIB interface, for converting an END driver that uses RFC 1213 to use the generic MIB interface, or for the RFC 2233 pulled method.

The pushed method of implementing RFC 2233 requires the device driver to call an API for every received frame or transmitted packet. This method has proven inappropriate for gigabit drivers because it includes substantial overhead that degrades performance. In most cases, it is also unnecessary because many gigabit devices capture most, if not all, the required information themselves. For these reasons, the pulled method was developed. In the pulled method, the driver provides the network stack with an API through which it can demand the current values in the hardware registers. Rather than the driver calling a MIB interface for each frame or transmitted packet, the stack periodically calls a driver API that provides statistical data captured on-demand in the hardware registers.

The pulled method can only be implemented on devices that provide hardware statistical capture registers. This feature is available for most gigabit devices. However, it is not guaranteed for all gigabit devices and is even more unlikely for 10/100 devices. Therefore, the pushed method must still be available as an option. However, *if* the pulled method *is* available, it should be used.

This document provides instructions for implementing both methods.

#### Pushed Method

This describes a generic facility capable of working transparently with RFC 1213 or RFC 2233.

The following generic API routines have been added to **endLib.c**. As a result the **endMibIfInit( )**, **mib2Init( )**, and **mib2ErrorAdd( )** routines are marked as obsolete.

**endM2Init( )**

Drivers should call **endM2Init( )** with the proper arguments in their *end***Load( )** routine.

The **endM2Init( )** routine determines if RFC 2233 is available or not, and sets a global flag accordingly. This needs to be done only once, but does not cause problems if done repeatedly.

The routine stores the physical address in the appropriate place (RFC 1213 or RFC 2233), initializes any required data structures, and does the equivalent work of **END_OBJ_READY**.

```
endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
          (u_char *) &enetAddr[0], 6, ETHERMTU, speed,
           IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);
```

**endM2Ioctl( )**

If a driver's **ioctl( )** is called with a **EIOCGMIB2** or **EIOCGMIB2233**, it must call **endM2Ioctl( )**.

**endM2Packet( )**

When a driver receives or sends a packet, encounters an error, or discards a packet, it must call **endM2Packet( )**.

```
endM2Packet(pEnd, pMBlk, counter)
```

Where counter is one of the following:

- **M2_PACKET_IN**
- **M2_PACKET_OUT**
- **M2_PACKET_IN_ERROR**
- **M2_PACKET_IN_DISCARD**
- **M2_PACKET_OUT_ERROR**
- **M2_PACKET_OUT_DISCARD**

In the **M2_PACKET_IN_ERROR** case, the **pMblk** can be **NULL**, in other cases, it is a valid pointer. The routine inspects the **mblk** to determine which counters to update.

**NOTE: endM2Packet( )** can pass a **NULL** for **pMblk** when it fails to obtain a tuple from **netBufLib**. In this case, it specifies that the **M2_PACKET_IN_ERROR** counter should be updated.

It is vital that all **endM2Packet( )** calls be located in such a place that their validity is guaranteed. That is, do not log a successful receipt or send of a packet until it is absolutely certain that the packet has been successfully

received or sent. Special care should be taken to ensure that all failure conditions are properly logged.

**endM2Free( )**
A driver must call **endM2Free( )** in its unload routine. This routine frees the appropriate structures (that is, any allocated by **endM2Init( )**).

#### Implementing the Generic MIB Pushed Method

The following instructions document the process of implementing the generic MIB pushed method. With these instructions, you can convert an old RFC 1213 MIB interface to use the generic MIB or, you can use these instructions to implement the generic MIB in a driver that does not have the RFC 1213 MIB interface already implemented. If the original driver does not already support RFC 1213, ignore the instructions to remove the RFC 1213 interface API.

1. In the driver *end***Load( )** routine, call **endM2Init( )**

   Initialize MIB-II entries (for RFC 2233 **ifXTable**)

   For example:

   ```
   endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
           (u_char *) &enetAddr[0], FEI_ADDR_LEN, ETHERMTU, speed,
           IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);
   ```

2. In the driver *end***Unload( )** routine, call **endM2Free( )**.

   ```
   endM2Free (pDrvCtrl);
   ```

3. Add the **EIOCGMIB2233** case in the **ioctl( )** routine.

   If the driver's **ioctl( )** is called with a **EIOCGMIB2** or **EIOCGMIB2233**, call **endM2Ioctl( )**.

   For example:

   ```
   /* New RFC 2233 mib2 interface */

   case EIOCGMIB2233:
   case EIOCGMIB2:

       endM2Ioctl (pDrvCtrl, cmd, datal);

       break;
   ```

4. Replace the old RFC 1213 interface API with the generic MIB interface API, then delete the old RFC 1213 interface API.

RFC 1213 Interface API:

The old RFC 1213 interface used the **END_ERR_ADD** macro for both updating packet counts and for counting error conditions.

a. Replace all instances of **END_ERR_ADD** calls. After an **END_ERR_ADD** instance is replaced, it can be deleted.

   For example:

   ```
   END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_UCAST, +1);
   ```

b. Replace the deleted RFC 1213 Interface.

   i. In the send and polling send routines, add the generic **mib2** counter update for outgoing packets.

      Send routine:

      ```
      endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_OUT);
      ```

      Polling send routine:

      ```
      endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_OUT);
      ```

   ii. In the receive and polling receive routines add the generic **mib2** counter update for incoming packets.

      Receive routine:

      ```
      endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_IN);
      ```

      Polling receive routine:

      ```
      endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_IN);
      ```

5. Log failure and error conditions.

Special care should be used to ensure that all failure conditions are properly logged.

All failure conditions are considered errors. However, there are two general classes of failure conditions. These can be either an error status returned by the device due to failure to accomplish a requested action, or the driver's inability to handle a packet due the lack of available resources.

In the case of device failure conditions, the conditions can be broken down further into errors only and errors with discards. This is determined by whether a failure causes packets to be dropped or not dropped. In the case where no packets are dropped, it is only an error. In the case where data is dropped, it is both an error and a discard. In almost all cases, it turns out that device errors are both an error and a discard.

If the driver received a packet that was corrupted at receipt then that would be regarded as only an error. However, in the case of the driver's inability to handle a perfectly good packet due to the lack of available resources, this is always both an error and a discard.

Example:

```
endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_IN_ERROR);
endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_IN_DISCARD);
```

### Pulled Method

The following instructions detail the implementation of the generic MIB pulled method. It is not anticipated that these instructions will be used with drivers that have already implemented the RFC 1213 MIB interface.

1. Add **END_IFDRVCONF** and **END_IFCOUNTERS** structures to the driver's control structure as follows:

```
END_IFDRVCONF endStatsConf;
END_IFCOUNTERS endStatsCounters;
} DRV_CTRL;
```

2. Declare a status dump routine.

```
LOCAL STATUS     gei82543EndStatsDump     (END_DEVICE *);
```

3. Modify the END driver load routine.

```
endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
         (u_char *) &enetAddr[0], 6, ETHERMTU, speed,
         IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);

bzero ((char *)&pDrvCtrl->endStatsCounters, sizeof(END_IFCOUNTERS));

pDrvCtrl->endStatsConf.ifPollInterval = sysClkRateGet();
pDrvCtrl->endStatsConf.ifEndObj = &pDrvCtrl->end;
pDrvCtrl->endStatsConf.ifWatchdog = NULL;
pDrvCtrl->endStatsConf.ifValidCounters = (END_IFINUCASTPKTS_VALID      |
                                          END_IFINMULTICASTPKTS_VALID  |
                                          END_IFINBROADCASTPKTS_VALID  |
                                          END_IFINOCTETS_VALID         |
                                          END_IFOUTOCTETS_VALID        |
                                          END_IFOUTUCASTPKTS_VALID     |
                                          END_IFOUTMULTICASTPKTS_VALID |
                                          END_IFOUTBROADCASTPKTS_VALID);
```

4. Modify the **ioctl( )** routine.

```
case EIOCGMIB2233:
case EIOCGMIB2:

    endM2Ioctl (pDrvCtrl, cmd, datal);
```

```
        break;

    case EIOCGPOLLCONF:
        if ((data == NULL))
            error = EINVAL;
        else
            *((END_IFDRVCONF **)data) = &pDrvCtrl->endStatsConf;
        break;

    case EIOCGPOLLSTATS:
        if ((data == NULL))
            error = EINVAL;
        else
            {
            error = gei82543EndStatsDump(pDrvCtrl);
            if (error == OK)
                *((END_IFCOUNTERS **)data) = &pDrvCtrl->endStatsCounters;
            }
        break;
```

5. Define an *xxx***EndStatsDump( )** routine.

   This routine dumps the register contents in the format expected by the MIB. The register set in a particular device may not exactly match the data set expected by the MIB. When this is the case, the *xxx***EndStatsDump( )** routine, if possible, performs what arithmetic is necessary to modify the device's registered data set to the MIB's expectations.

In the following example, the device counts multicast and broadcast packets and all incoming packets but does not specifically count unicast packets. The *xxx***EndStatsDump( )** routine calculates the unicast value by subtracting the multicast and broadcast values from the count of all incoming packets.

Example:

```
/*******************************************************************************
*
* gei82543EndStatsDump - Dump statistic registers for MIB2 (RFC 2233)
*
* This routine dumps statistic registers for MIB2 update
*
* RETURNS: OK
*/

LOCAL STATUS gei82543EndStatsDump
    (
    END_DEVICE *     pDrvCtrl    /* device receiving command */
    )
    {
    END_IFCOUNTERS *    pEndStatsCounters;
    UINT32              tmp;

    pEndStatsCounters = &pDrvCtrl->endStatsCounters;
```

```
/*
 * Get number of RX'ed octets
 * Note: the octet counts are 64-bit quantities saved in two
 * 32-bit registers. Reading the high word clears the count,
 * so we have to read the low word first.
 */

GEI_READ_REG(INTEL_82543GC_GORL, tmp);
pEndStatsCounters->ifInOctets = tmp;
GEI_READ_REG(INTEL_82543GC_GORH, tmp);
pEndStatsCounters->ifInOctets |= (unsigned long long)tmp << 32;

/* Get number of TX'ed octets */

GEI_READ_REG(INTEL_82543GC_GOTL, tmp);
pEndStatsCounters->ifOutOctets = tmp;
GEI_READ_REG(INTEL_82543GC_GOTH, tmp);
pEndStatsCounters->ifOutOctets |= (unsigned long long)tmp << 32;

/* Get RX'ed unicasts, broadcasts, multicasts */

GEI_READ_REG(INTEL_82543GC_GPRC, tmp);
pEndStatsCounters->ifInUcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_BPRC, tmp);
pEndStatsCounters->ifInBroadcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_MPRC, tmp);
pEndStatsCounters->ifInMulticastPkts = tmp;
pEndStatsCounters->ifInUcastPkts -=
                                  (pEndStatsCounters->ifInMulticastPkts +
                                   pEndStatsCounters->ifInBroadcastPkts);
/* Get TX'ed unicasts, broadcasts, multicasts */

GEI_READ_REG(INTEL_82543GC_GPTC, tmp);
pEndStatsCounters->ifOutUcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_BPTC, tmp);
pEndStatsCounters->ifOutBroadcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_MPTC, tmp);
pEndStatsCounters->ifOutMulticastPkts = tmp;
pEndStatsCounters->ifOutUcastPkts -=
                                  (pEndStatsCounters->ifOutMulticastPkts +
                                   pEndStatsCounters->ifOutBroadcastPkts);

return (OK);
}
```

6. Modify the unload routine.

```
/* Free MIB-II entries */
endM2Free(DRV_CTRL*);
```

# 4 SCSI Drivers


4.1 Introduction 93

4.2 SCSI Overview 94

4.3 SCSI BSP Interface 132

4.4 The SCSI Driver Development Process 135

4.5 Common SCSI Driver Development Issues 135


## 4.1 Introduction

**NOTE:** The information in the chapter is provided for reference purposes only. You should use this information to maintain existing SCSI driver code. If you want to develop a new driver, see *VxWorks Device Driver Developers Guide, Volume 1*.

The VxWorks SCSI-2 subsystem consists of the following components:

- SCSI libraries, an architecture-independent component
- SCSI controller driver, an architecture-specific component
- SCSI-2 subsystem initialization code, a board-specific component

You must first understand the basic functionality of each of these components before you can extend the functionality of the SCSI libraries or add new SCSI

controller drivers. To help you gain that understanding, this chapter describes the general layout of the various SCSI modules, discusses the internals of the SCSI libraries (and their programming interface with the SCSI controller drivers), and describes the process of developing a controller-specific SCSI driver.

When a VxWorks task requests SCSI service by invoking a SCSI library routine such as **scsiInquiry( )**. Since we are assuming a SCSI-2 configuration, first the **scsi2Inquiry( )** routine is invoked which in turn invokes **scsiTransact( )** (see *Forming SCSI Commands*, p.97). **scsiTransact( )** invokes **scsiCommand( )**, the routine that allocates a SCSI thread, executes the thread, and then deletes it.

The execution of the thread via **scsiThreadExecute( )** causes the SCSI manager to be informed of a new thread to execute, and subsequent blocking of that VxWorks task on a message queue until a response has been received. This is the boundary where a VxWorks task is blocked and the SCSI manager is awakened to start the execution of a new thread as well as management of any other threads that it may be dealing with.

After the SCSI thread has executed and has received a response, the calling VxWorks task is unblocked and eventually the SCSI thread associated with that task is deleted.

For information on the interface between the I/O system and the SCSI libraries, including configuring SCSI peripheral devices within VxWorks, see the *VxWorks Kernel Programmer's Guide: I/O System*.

**NOTE:** In this chapter, the term *SCSI* refers to SCSI-2 in all cases. The SCSI library interfaces and SCSI controller drivers described in this chapter refer to SCSI-2 only. VxWorks offers only limited support for SCSI-1. Eventually, Wind River will eliminate all SCSI-1 support from VxWorks.

## 4.2 SCSI Overview

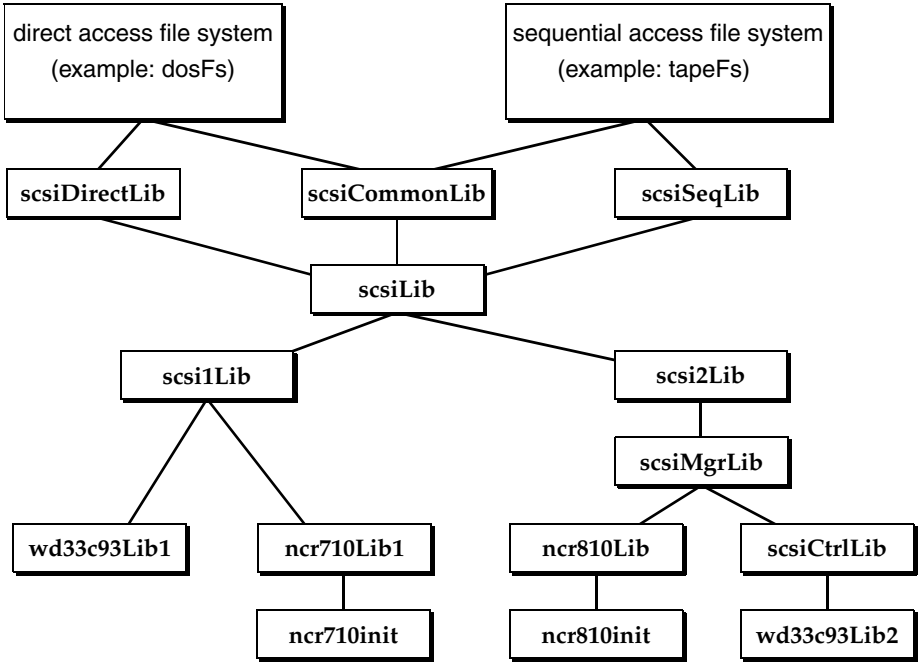
This section describes the relationships between various SCSI modules, introduces the different SCSI objects and data structures, and tells you how to form SCSI commands.

### 4.2.1 Layout of SCSI Modules

Figure 4-1 shows all the SCSI library modules and the relationship between them and several typical drivers. The SCSI libraries contain a variety of data structures. The important data structures and their relationships are described in the following subsections. The general design of the data structures is object-oriented; data structures represent real and abstract SCSI objects such as peripheral devices, controllers, and block devices.



Figure 4-1 **Layout of SCSI Modules**




#### SCSI Objects and Data Structures

Figure 4-2 illustrates the relationship between the various physical and logical SCSI objects and the corresponding data structures.

Figure 4-2 **Relationship of SCSI Devices and Data Structures**

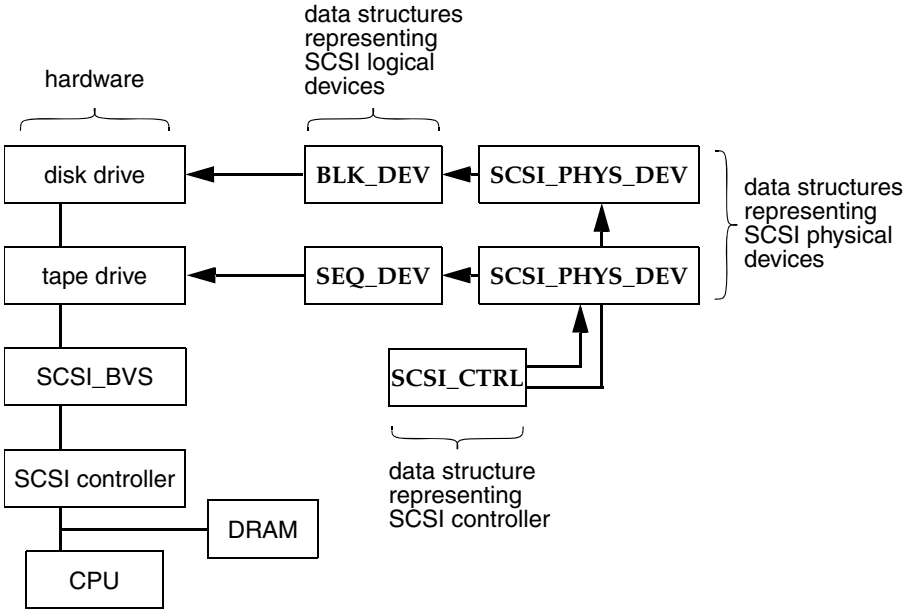



Figure 4-3 illustrates the contents of these data structures and their relationships in more detail.

**SCSI_CTRL**
This structure contains a list of all physical devices and all allocated SCSI threads.

**SCSI_THREAD**
Each thread is represented by a dynamic data structure, which is manipulated at various levels in **scsi2Lib**, **scsiMgrLib**, and the device drivers. It contains a **SCSI_TRANSACTION** and the rest of the thread-state information.

**SCSI_TRANSACTION**
Each SCSI command from the I/O system is translated into one of these structures, which consists of a SCSI command descriptor block plus all the required pointer addresses.

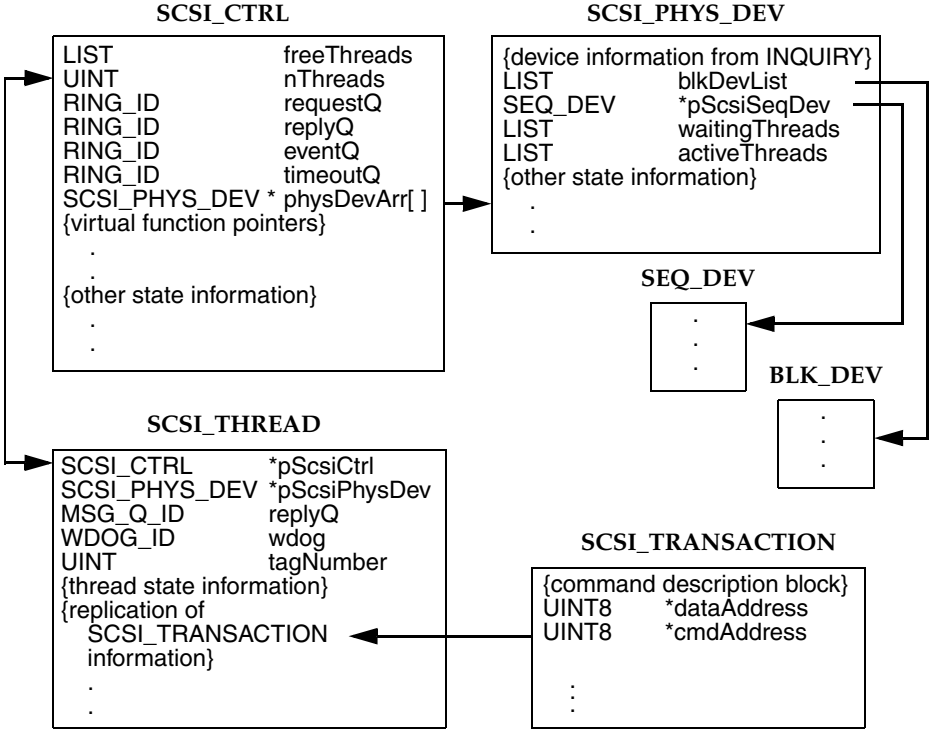
**SCSI_PHYS_DEV**
This structure contains information about available logical devices plus information about the various threads.

**SEQ_DEV**
This structure represents a sequential logical device such as a tape drive.

**BLK_DEV**
This structure represents a block device such as a disk drive.



Figure 4-3 **Controller- and Driver-Specific Data Structures**




### Forming SCSI Commands

Within the SCSI libraries, the SCSI commands all work in a similar fashion. All information needed by the command is delivered by passing in appropriate parameters. The command first builds a SCSI command descriptor block with pointers to all required data and stores the block in a **SCSI_TRANSACTION** structure. The command then calls the **scsiTransact( )** routine, passing it the structures **SCSI_TRANSACTION** and **SCSI_PHYS_DEV**.

The **scsiTransact( )** routine is the general routine in **scsi2Lib** that handles processing of all SCSI commands originating in **scsiDirectLib**, **scsiCommonLib**,

and **scsiSeqLib**. This paradigm should be used to extend SCSI library support to other device classes (**scsi***XXX***Lib**).

```
STATUS scsiXxxCmd
    (
    char * buf
    SCSI_PHYS_DEV * pScsiPhysDev
    )
```

## 4.2.2 The VxWorks OS Interface

This section discusses how SCSI drivers interface with the VxWorks operating system.

#### Libraries

This section describes the following libraries:

- The SCSI Manager **(scsiMgrLib)**
- SCSI Controller Library **(scsiCtrlLib)**
- SCSI Direct Access Library **(scsiDirectLib)**
- SCSI Sequential Access Library **(scsiSeqLib)**
- SCSI Common Access Library **(scsiCommonLib)**

This section ends with a brief discussion of how VxWorks typically handles the execution of a SCSI command.

#### SCSI Manager (scsiMgrLib)

The SCSI manager functions as a task within VxWorks. There is one SCSI manager per SCSI controller, and it is responsible for managing all SCSI interaction between VxWorks tasks and the SCSI controller. Any number of VxWorks tasks can request services from SCSI peripheral devices. The SCSI bus is a shared critical resource which requires multitasking support and synchronization.

For the sake of performance and efficiency, the SCSI manager controls all the SCSI traffic within the operating system. SCSI traffic includes requests for SCSI services by VxWorks tasks. These requests are asynchronous events from the SCSI bus and include SCSI reconnects, SCSI connection time-outs, and SCSI responses to requests by VxWorks tasks. This work flow is managed by SCSI threads, which are

SCSI-library-specific abstractions. A SCSI thread is assigned to each unit of SCSI work. In other words, one SCSI thread is assigned per SCSI request.

Each SCSI thread is created in the context of the calling VxWorks task. The thread is managed by the SCSI manager, while the calling VxWorks task remains blocked. When the SCSI thread completes, the VxWorks task is unblocked and the SCSI thread is deleted.



A SCSI thread has its own context or state variables, which are manipulated by the SCSI libraries and the controller driver. A maximum of one SCSI thread can be executing at any one time. In addition to managing the SCSI-thread state information, the SCSI manager is responsible for scheduling these SCSI threads.

When there are multiple threads in existence, the different threads can be in various states representing different requirements. A SCSI thread can represent a new request for service, a connection time-out, a completion of service, or an event from the SCSI bus. As requests for service are submitted to the SCSI manager by VxWorks tasks, the associated threads must be processed based on priority or on a first-come-first-serves basis if their priority is the same.

When multiple threads are eligible for activation, the SCSI manager follows a strict hierarchy of processing. Asynchronous bus events have the highest priority and are processed before any other type of SCSI thread. The order of processing is: events, time-outs, requests, and finally responses. The SCSI manager handles any race condition that develops between activation of a request and the asynchronous occurrence of an event from the SCSI bus.

Once an appropriate SCSI thread is selected for execution, the SCSI manager dispatches that thread and actual execution is handled by the controller-specific driver.

#### Limitations

The SCSI manager uses standard VxWorks ring buffers to manage SCSI requests. Using ring buffers is fast and efficient. The amount of SCSI work that can be queued depends upon the size of the allocated ring buffers. The SCSI manager also has some limitations. For example:

- the maximum number of threads allowed (**scsiMaxNumThreads**)
- the maximum number of SCSI requests from VxWorks tasks that can be put on the SCSI manager's request queue (**scsiMgrRequestQSize**)
- the maximum number of SCSI bus events that can be put on the SCSI manager's event queue (**scsiMgrEventQSize**)
- the maximum number of replies that can be put on the reply queue (**scsiMgrReplyQSize**)

- the maximum number of time-outs that can be put on the time-out queue (**scsiMgrTimeoutQSize**)
- time-out values.

#### Configuration

It is possible to tune the size of the ring buffers and the number of SCSI threads to optimize a specific environment. In most cases, however, the default values are sufficient. These parameters—**scsiMaxNumThreads**, **scsiMgrRequestQSize**, **scsiMgrReplyQSize**, **scsiMgrEventQSize**, **scsiMgrTimeoutQSize**—are defined as global variables within the SCSI library and are assigned default values defined in **scsiLib.h**. These values can be reassigned in the BSP routine **sysScsiInit( )** prior to the invocation of the driver's *xxx***CtrlInit( )** routine. Then when **scsiCtrlInit( )**is invoked by the driver's *xxx***CtrlInit( )**routine, the new parameters are used for data structure allocation.

The name, priority, and stack size of the **scsiMgr** task can also be customized from the controller driver's *xxx***CtrlCreate( )** routine. Defaults are provided in **scsiLib.h**. For example, the default task name **SCSI_DEF_TASK_NAME** is **tScsiTask**, the default priority, **SCSI_DEF_TASK_PRIORITY**, is 5, and the default stack size, **SCSI_DEF_TASK_STACK_SIZE**, is 4000.

**NOTE:** The larger the number of expected VxWorks SCSI tasks, the larger the stack space required. Thought should be given to the stack size parameter when customizing the SCSI manager.

### SCSI Controller Library (scsiCtrlLib)

The SCSI controller library is designed for the older generation of SCSI-2 controllers that require the protocol state machine (and transitions) to be handled by a higher level of software.These basic SCSI controller drivers (those that need to use the SCSI state machine provided by the SCSI library) use the SCSI controller library. More advanced SCSI controllers allow such protocol state machines to be implemented at the SCSI controller level. This significantly reduces the number of SCSI interrupts to the CPU per I/O process which improves performance.

There is a well defined interface between the SCSI libraries and the controller driver of such drivers, and this interface is defined in *Driver Programming Interface*, p.101.

### SCSI Direct Access Library (scsiDirectLib)

The SCSI direct access library **scsiDirectLib** encapsulates all the routines that implement the SCSI direct access commands as defined in the *SCSI ANSI*

*Specification I*. In addition to all the direct access commands, **scsiDirectLib** provides the routines that supply the **BLK_DEV** abstraction for SCSI direct access peripheral devices.

#### SCSI Sequential Access Library (scsiSeqLib)

The SCSI sequential access library **scsiSeqLib** provides all the routines that implement the mandatory SCSI sequential access commands as defined in the *SCSI ANSI Specification I*. Some optional features are also implemented. Routines that manipulate the **SEQ_DEV** abstraction are also supplied in this library.

#### SCSI Common Access Library (scsiCommonLib)

SCSI commands that are common to all SCSI peripheral device types are provided in the common access library. These commands are described in the *SCSI ANSI Specification I*. The programming interface to such commands can be found in the relevant reference entries or by looking at the header file **scsi2Lib.h**.

### Driver Programming Interface

To better explain the interface between the controller driver and the SCSI libraries for the two types of SCSI controllers (basic and advanced), this section discusses each type of driver separately. A skeletal driver is provided along with the programming interface between the SCSI libraries and the controller driver. The controller driver routines provide all the hardware register accesses and controller-specific functionality. For the sake of simplicity, such accesses and controller-specific information have not been shown. It is the purpose of the template drivers to show the overall structure and programming interface between the driver, the SCSI libraries, and the BSP.

#### Basic SCSI Controller Driver

This section presents the basic programming interface SCSI controller and the SCSI libraries. Following that description, this section presents a template you should use when writing your own SCSI controller driver.

#### The Programming Interface

A well-defined programming interface exists between the controller driver of any basic SCSI controller and the SCSI libraries. Every basic controller driver must provide the following routines to the SCSI libraries:

*xxx***DevSelect( )**
This routine selects a SCSI peripheral device with the attention (**ATN**) signal asserted.

*xxx***InfoXfer( )**
All information transfer phases are handled by this routine, including the **DATA_IN**, **DATA_OUT**, **MSG_IN**, **MSG_OUT**, and **STATUS** phases.

*xxx***XferParamsQuery( )**
This routine updates the synchronous data transfer parameters to match the capabilities of the driver and returns the optimal synchronous offset and period.

*xxx***XferParamsSet( )**
This routine sets the synchronous data transfer parameters on the SCSI controller.

*xxx***BusControl( )**
This routine controls some of the SCSI bus lines from the controller. This routine must reset the SCSI bus, assert **ATN**, or negate **ACK**.

Similarly, the controller driver invokes the following routines in order to get SCSI library services:

**scsiCtrlInit( )**
This routine initializes the SCSI library data structures. It is called only once per SCSI controller.

**scsiMgrEventNotify( )**
This routine notifies the SCSI manager of a SCSI event that has occurred. Events are defined in **scsi2Lib.h**. However, more events can be defined by the controller driver, and events can also be bundled by the driver. In this case, the **SCSI_CTRL** field **scsiEventProc** must be set to this driver-specific routine during driver initialization.

### A Template Driver

The following example shows a template for a basic SCSI controller driver, without any specific hardware constraints. The basic structure of the driver is like any other VxWorks driver. The main routines consist of the following:

- A *xxx***CtrlCreate( )** routine, that is invoked from the BSP routine **sysScsiInit( )** located in the BSP file **sysScsi.c**.
- An ISR called *xxx***Intr( )** that handles all the interrupts, deciphers what SCSI event has occurred, and passes that event information to the SCSI manager via the **scsiMgrEventNotify( )** routine.

The SCSI libraries instruct the driver via the *xxx***DevSelect( )** and *xxx***InfoXfer( )** routines, and the controller driver communicates back to the libraries by means of the **scsiMgrEventNotify( )** routine.

Example 4-1 **Basic SCSI Controller Driver**

```
/* xxxLib.c - XXX SCSI-Bus Interface Controller library (SCSI-2) */

/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01a,12sep96,dds  written
*/

/*
DESCRIPTION
This library contains part of the I/O driver for the XXX family of SCSI-2
Bus Interface Controllers (SBIC).  It is designed to work with scsi2Lib.
The driver routines in this library depend on the SCSI-2 ANSI specification;
for general driver routines and for overall SBIC documentation, see xxxLib.

INCLUDE FILES
xxx.h

SEE ALSO: scsiLib, scsi2Lib,
the VxWorks programmer's guides
*/

#include "vxWorks.h"
#include "drv/scsi/xxx.h"


typedef XXX_SCSI_CTRL SBIC; /* SBIC: SCSI Bus Interface Controller struct */
```

```
/* globals */

int xxxXferDoneSemOptions  = SEM_Q_PRIORITY;
char *xxxScsiTaskName      = SCSI_DEF_TASK_NAME;

IMPORT SCSI_CTRL *pSysScsiCtrl;

/*******************************************************************************
* xxxCtrlCreate - create and partially initialize a SCSI controller structure
*
* This routine creates a SCSI controller data structure and must be called
* before using a SCSI controller chip.  It should be called once and only
* once for a specified SCSI controller. Since it allocates memory for a
* structure needed by all routines in xxxLib, it must be called before
* any other routines in the library.
* After calling this routine, at least one call to xxxCtrlInit() should
* be made before any SCSI transaction is initiated using the SCSI controller.
*
* RETURNS: A pointer to the SCSI controller structure, or NULL if memory is
* insufficient or parameters are invalid.
*/

XXX_SCSI_CTRL *xxxCtrlCreate
    (
    FAST UINT8 *sbicBaseAdrs,     /* base address of the SBIC              */
    int         regOffset,        /* address offset between SBIC registers */
    UINT        clkPeriod,        /* period of the SBIC clock (nsec)       */
    FUNCPTR     sysScsiBusReset,  /* function to reset SCSI bus            */
    int         sysScsiResetArg,  /* argument to pass to above function    */
    UINT        sysScsiDmaMaxBytes,  /* maximum byte count using DMA       */
    FUNCPTR     sysScsiDmaStart,  /* function to start SCSI DMA transfer   */
    FUNCPTR     sysScsiDmaAbort,  /* function to abort SCSI DMA transfer   */
    int         sysScsiDmaArg     /* argument to pass to above functions   */
    )
    {
    FAST SBIC *pSbic;             /* ptr to SBIC info */

    /* calloc the controller info structure; return NULL if unable */
    pSbic = (SBIC *) calloc (1, sizeof (SBIC))

    /*
     *  Set up sizes of event and thread structures. Must be done before
     *  calling "scsiCtrlInit()".
     */

    /*  fill in driver-specific routines for scsiLib interface */

    pSbic->scsiCtrl.scsiDevSelect       = xxxDevSelect;
    pSbic->scsiCtrl.scsiInfoXfer        = xxxInfoXfer;
    pSbic->scsiCtrl.scsiXferParamsQuery = xxxXferParamsQuery;
    pSbic->scsiCtrl.scsiXferParamsSet   = (FUNCPTR)xxxXferParamsSet;

    /* Fill in driver specific variables for scsiLib interface */

    pSbic->scsiCtrl.maxBytesPerXfer = sysScsiDmaMaxBytes;
```

```
    /* fill in generic SCSI info for this controller */

    xxxCtrlInit (&pSbic->scsiCtrl);

    /* initialize SBIC info transfer synchronization semaphore */

    if (semBInit (&pSbic->xferDoneSem, xxxXferDoneSemOptions, SEM_EMPTY)
                == ERROR)
        {
        (void) free ((char *) pSbic);
        return ((XXX_SCSI_CTRL *) NULL);
        }

    /* initialize state variables */

    /* fill in board-specific SCSI bus reset and DMA xfer routines */

    /* spawn SCSI manager - use generic code from "scsiLib.c" */

    pSbic->scsiCtrl.scsiMgrId = taskSpawn (xxxTaskName,
                                           xxxTaskPriority,
                                           xxxTaskOptions,
                                           xxxTaskStackSize,
                                           (FUNCPTR) scsiMgr,
                                           (int) pSbic,
                                           0, 0, 0, 0, 0, 0, 0, 0, 0);

    return (pSbic);
    }

/*******************************************************************************
* xxxCtrlInit - initialize a SCSI controller structure
*
* After a SCSI controller structure is created with xxxCtrlCreate, but
* before using the SCSI controller, it must be initialized by calling this
* routine.
* It may be called more than once if desired.  However, it should only be
* called while there is no activity on the SCSI interface.
*
* RETURNS: OK, or ERROR if out-of-range parameter(s).
*/

LOCAL STATUS xxxCtrlInit
    (
    FAST SBIC *pSbic,           /* ptr to SBIC info */
    FAST int  scsiCtrlBusId,    /* SCSI bus ID of this SBIC */
    FAST UINT defaultSelTimeOut /* default dev. select timeout (microsec) */
    )
    {
    pSbic->scsiCtrl.scsiCtrlBusId = scsiCtrlBusId;
```

```
    /* initialize the SBIC hardware */

    xxxHwInit (pSbic);

    return (OK);
    }

/*******************************************************************************
* xxxHwInit - initialize the SCSI controller to a known state
*
* This routine puts the SCSI controller into a known quiescent state.  It
* does not reset the SCSI bus (and any other devices thereon).
*/

LOCAL void xxxHwInit
    (
    SBIC *pSbic                         /* ptr to an SBIC structure */
    )
    {
    /*
     * Initialize the SCSI controller hardware registers and place the
     * chip in a known quiescent state
     */
    }

/*******************************************************************************
* xxxDevSelect - attempt to select a SCSI device
*
* RETURNS: OK (no error conditions)
*/

LOCAL STATUS xxxDevSelect
    (
    SCSI_CTRL *pScsiCtrl,       /* ptr to SCSI controller info      */
    int        devBusId,        /* SCSI bus ID of device to select  */
    UINT       selTimeOut,      /* select t-o period (usec)         */
    UINT8     *msgBuf,          /* ptr to identification message    */
    UINT       msgLen           /* maximum number of message bytes  */
    )
    {
    int        lockKey;         /* saved interrupt lock key         */

    lockKey = intLock ();

    /* Select  device */

    intUnlock (lockKey);
    }

/*******************************************************************************
* xxxXferParamsQuery - get (synchronous) transfer parameters
*
* Updates the synchronous transfer parameters suggested in the call to match
* the SCSI controller's capabilities.  Transfer period is in SCSI units
* (multiples of 4 ns).
*
```

```
* RETURNS: OK
*/

LOCAL STATUS xxxXferParamsQuery
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to SBIC info             */
    UINT8     *pOffset,                 /* max REQ/ACK offset  [in/out] */
    UINT8     *pPeriod                  /* min transfer period [in/out] */
    )
    {
    /* read offset and period values */

    return (OK);
    }

/*******************************************************************************
*
* xxxXferParamsSet - set transfer parameters
*
* Programs the SCSI controller to use the specified transfer parameters. An
* offset of zero specifies asynchronous transfer (period is then irrelevant).
*
* RETURNS: OK if transfer parameters are OK, else ERROR.
*/

LOCAL STATUS xxxXferParamsSet
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to SBIC info       */
    UINT8      offset,                  /* max REQ/ACK offset     */
    UINT8      period                   /* min transfer period    */
    )
    {
    /* set the appropriate SCSI controller registers */

    return (OK);
    }

/*******************************************************************************
* xxxInfoXfer - transfer information bytes to/from target via SCSI bus
*
* Executes a "Transfer Info" command to read (write) bytes from (to) the
* SCSI bus.  If the transfer phase is DATA IN or DATA OUT and there is a
* DMA routine available, DMA is used - otherwise it's a tight programmed
* i/o loop.
*
* RETURNS: Number of bytes transferred across SCSI bus, or ERROR.
*/

LOCAL int xxxInfoXfer
    (
    FAST SCSI_CTRL *pScsiCtrl,       /* ptr to SCSI controller info       */
    int             phase,           /* SCSI phase being transferred      */
    FAST UINT8     *pBuf,            /* ptr to byte buffer for i/o        */
    UINT            bufLength        /* number of bytes to be transferred */
    )
    {
```

```
    pSbic = (SBIC *) pScsiCtrl;

    /* Handle phase changes */

    /* Start DMA, if used, or programmed i/o loop to transfer data */

    /* Wait for transfer to complete: find out how many bytes transferred */

    semTake (&pSbic->xferDoneSem, WAIT_FOREVER);

    /*
     * If there are bytes left to be transferred return ERROR
     * If DMA is used for transfer do a SCSI DMA Abort
     */

    xxxXferCountGet (pSbic, &bytesLeft);

    return (bufLength - bytesLeft);
    }

/*******************************************************************************
* xxxXferCountSet - load the SCSI controller transfer counter with count.
*
* RETURNS: OK if count is in range 0 - 0xffffff, otherwise ERROR.
*
*/

LOCAL STATUS xxxXferCountSet
    (
    FAST SBIC *pSbic,                   /* ptr to SBIC info */
    FAST UINT count                     /* count value to load */
    )
    {
    /* set the appropriate SCSI controller registers */
    }

/*******************************************************************************
* xxxXferCountGet - fetch the SCSI controller transfer count
*
* The value of the transfer counter is copied to *pCount.
*
*/

LOCAL void xxxXferCountGet
    (
    FAST SBIC *pSbic,                   /* ptr to SBIC info */
    FAST UINT *pCount                   /* ptr to returned value */
    )
    {
    /* read the appropriate SCSI controller registers */
    }

/*******************************************************************************
* xxxCommand - write a command code to the SCSI controller Command Register
*
*/
```

```
LOCAL void xxxCommand
    (
    SBIC *pSbic,                        /* ptr to SBIC info */
    UINT8 cmdCode                       /* new command code */
    )
    {
    /* set the appropriate SCSI controller registers */
    }

/*******************************************************************************
* xxxIntr - interrupt service routine for the SCSI controller
*
*/
LOCAL void xxxIntr
    (
    SBIC *pSbic                         /* ptr to SBIC info */
    )
    {
    SCSI_EVENT event;

    /* Check the SCSI status. Handle state transitions */

    switch (scsiStatus)
        {
        ...

        /* the list of event types is defined is scsi2Lib.h */

        case ...

            event.type  = SCSI_EVENT_XFER_REQUEST;
            event.phase = busPhase;
            break;

        case ...
        }


    /* Synchronize with task-level code */

    semGive (&pSbic->xferDoneSem);

    /* Post event to SCSI manager for further processing */
    scsiMgrEventNotify ((SCSI_CTRL *)pSbic, &event, sizeof (event));
    }

/*******************************************************************************
* xxxRegRead - Get the contents of a specified SCSI controller register
*/

LOCAL void xxxRegRead
    (
    SBIC *pSbic,                        /* ptr to an SBIC structure    */
    UINT8 regAdrs,                      /* address of register to read */
    int  *pDatum                        /* buffer for return value     */
```

```
    )
    {
    /* read the appropriate SCSI controller registers */
    }

/*************************************************************************
* xxxRegWrite - write a value to a specified SCSI controller register
*
*/

LOCAL void xxxRegWrite
    (
    SBIC *pSbic,                        /* ptr to an SBIC structure     */
    UINT8 regAdrs,                      /* address of register to write */
    UINT8 datum                         /* value to be written          */
    )
    {
    /* write the appropriate SCSI controller registers */
    }
```

### Advanced SCSI Controller Driver

The advanced SCSI controller incorporates all the low-level state machine routines within the driver. This functionality replaces that provided by **scsiCtrlLib**. Most advanced SCSI controllers have their own SCSI I/O processor which enhances performance by managing all the low-level activities on the SCSI bus, such as phase changes and DMA data transfers. Usually the instructions to the I/O processor are machine language instructions which are written in a higher level assembly language and compiled into machine instructions. These machine instructions reside in the main DRAM area and are fetched by the I/O processor from DRAM by using a SCSI program counter and some form of indirect addressing.

In the case of advanced SCSI controllers, there is usually additional event information described in a driver-specific structure such as *XXX*_**EVENT** (where *XXX* refers to the SCSI driver module prefix). Many thread management routines are part of the controller driver, which is not true of the basic SCSI controller drivers.

#### The Programming Interface

The programming interface between the advanced SCSI controller driver and the SCSI libraries consists of routines that must be supplied by the driver and library routines which are invoked by the driver. The driver routines are not required to conform to the naming convention used here, because the routines are accessed by means of function pointers which are set in the *xxx***CtrlCreate( )** routine. However,

this naming convention is recommended. The routines (or equivalents) that the driver must supply are:

*xxx***EventProc( )**[1]
This routine is invoked by the SCSI manager to parse events and take appropriate action.

*xxx***ThreadInit( )**
This routine initializes the SCSI thread structures and adds any driver-specific initialization required beyond what is provided by **scsiThreadInit( )**.

*xxx***ThreadActivate( )**
This routine activates a SCSI connection, setting the appropriate thread context in the **SCSI_THREAD** data structure and setting all the controller registers with the appropriate values. It may call other driver routines as well as SCSI library routines.

*xxx***ThreadAbort( )**
If the thread is not actually connected, this routine does nothing. If the thread is connected, it sends an ABORT TAG message which causes the SCSI target to disconnect.

*xxx***BusControl( )**
This routine controls some of the SCSI bus lines from the controller. This routine must reset the SCSI bus, assert **ATN**, or negate **ACK**.

*xxx***XferParamsQuery( )**
This routine updates the synchronous data transfer parameters to match the capabilities of the driver and returns the optimal synchronous offset and period.

*xxx***XferParamsSet( )**
This routine sets the synchronous data transfer parameters on the SCSI controller.

*xxx***WideXferParamsQuery( )**
This routine updates the wide data transfer parameters in the call to match those of the SCSI controller.

*xxx***WideXferParamsSet( )**
This routine sets the wide data transfer parameters on the SCSI controller.

The advanced controller driver also uses many of the facilities provided by the SCSI libraries. All the routines invoked by the SCSI controller library can also be

1. The *xxx* in the routine name is just a place holder for whatever prefix you assign to your SCSI driver module.

invoked by the driver. Examining the SCSI controller library and the header file **scsi2Lib.h** shows all the routines available for the controller driver. The following list is a typical but not exhaustive list of routines that can be invoked by the driver:

**scsiCtrlInit( )**
This routine initializes the SCSI library data structures. It is called only once per SCSI controller.

**scsiMgrEventNotify( )**
This routine notifies the SCSI manager of an event that occurred on the SCSI bus.

**scsiWideXferNegotiate( )**
This routine initiates or continues wide data transfer negotiation. See the relevant reference entries and **scsi2Lib.h** for more details. It is typically invoked from the *xxx***ThreadActivate( )** routine.

**scsiSyncXferNegotiate( )**
This routine initiates or continues synchronous data transfer negotiations. See the relevant reference entries and **scsi2Lib.h** for more details. It is typically invoked from the *xxx***ThreadActivate( )** routine.

**scsiMgrCtrlEvent( )**
This routine sends an event to the SCSI controller state machine. It is usually called by the driver *xxx***EventProc( )** routine after a selection, re-selection, or disconnection.

**scsiMgrBusReset( )**
This routine resets all physical devices in the SCSI library upon a bus-initiated reset. It is typically invoked from *xxx***EventProc( )**.

**scsiMgrThreadEvent( )**
This routine sends an event to the thread state machine. It is called by the thread management routines within the driver; the entry point to the thread routines is by way of *xxx***EventProc( )**. In general, *xxx***EventProc( )** is the general routine which calls other driver-specific thread-management routines. For a better understanding, look at the advanced SCSI controller driver template and also examine an actual driver.

**scsiMsgOutComplete( )**
This routine performs post-processing after a SCSI message out has been sent. It is also invoked from the driver thread management routines.

**scsiMsgInComplete( )**
This routine performs post-processing after a SCSI message in is received. It is invoked from the driver thread management routines.

**scsiMsgOutReject( )**
This routine performs post-processing when an outgoing message has been rejected.

**scsiIdentMsgParse( )**
This routine parses an incoming identify message when VxWorks has been selected or reselected.

**scsiIdentMsgBuild( )**
This routine builds an identify message in the caller's buffer.

**scsiCacheSnoopEnable( )**
This routine informs the library that hardware cache snooping is enabled and that it is unnecessary to call cache-specific routines.

**scsiCacheSnoopDisable( )**
This routine informs the library that hardware snooping has been disabled or does not exist and that the library must perform cache coherency.

**scsiCacheSynchronize( )**
This routine is called by the driver for all cache-coherency needs.

**scsiThreadInit( )**
This routine performs general thread initialization; it is invoked by the driver *xxx***ThreadInit( )** routine.

Example 4-2 provides an advanced SCSI controller driver template and Example 4-3 shows a SCSI I/O processor assembly language template. These examples show how such drivers may be structured. Many details are not included in the templates; these templates simply serve to provide a high-level picture of what is involved. Once the basic structure of the template is understood, examining an actual advanced controller driver clarifies the issues involved, especially thread management.

Example 4-2 **Advanced Controller Driver Example**

```
/* xxxLib.c - XXX SCSI I/O Processor (SIOP) library */

/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01g,19aug96,dds  written
*/
```

```
/*
DESCRIPTION
This is the I/O driver for the XXX SCSI I/O Processor (SIOP).
It is designed to work with scsiLib and scsi2Lib.  This driver
runs in conjunction with a script program for the XXX controller.
These scripts use DMA transfers for all data, messages and status.
This driver supports cache functions through scsi2Lib.

USER-CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O
system.  The following routines must be called directly:  xxxCtrlCreate()
to create a controller structure, and xxxCtrlInit() to initialize it.
The XXX SCSI Controller's hardware registers need to be configured according
to the hardware implementation.  If the default configuration is not proper,
the routine xxxSetHwRegister() should be used to properly configure
the registers.

\INTERNAL
This driver supports multiple initiators, disconnect/reconnect, tagged
command queueing, synchronous data transfer and wide data transfer protocols.
In general, the SCSI system and this driver automatically choose the
best combination of these features to suit the target devices used.
However, the default choices may be over-ridden by using the function
"scsiTargetOptionsSet()" (see scsi2Lib).

There are debug variables to trace events in the driver.
<scsiDebug> scsiLib debug variable, trace event in scsiLib, xxxScsiPhase(),
and xxxTransact().
<scsiIntsDebug> prints interrupt information.

INCLUDE FILES
xxx.h, xxxScript.h and scsiLib.h
*/

#define  INCLUDE_SCSI2
#include "vxWorks.h"
#include "memLib.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "logLib.h"
#include "semLib.h"
#include "intLib.h"
#include "errnoLib.h"
#include "cacheLib.h"
#include "taskLib.h"
#include "drv/scsi/xxx.h"
#include "drv/scsi/xxxScript.h"

/* defines */

typedef XXX_SCSI_CTRL SIOP;
```

```
/* Configurable options */

int   xxxSingleStepSemOptions = SEM_Q_PRIORITY;
char *xxxScsiTaskName         = SCSI_DEF_TASK_NAME;
int   xxxScsiTaskOptions      = SCSI_DEF_TASK_OPTIONS;
int   xxxScsiTaskPriority     = SCSI_DEF_TASK_PRIORITY;
int   xxxScsiTaskStackSize    = SCSI_DEF_TASK_STACK_SIZE;

/*******************************************************************************
*
* xxxCtrlCreate - create a control structure for the XXX SCSI controller
*
* This routine creates a SCSI controller data structure and must be called
* before using a SCSI controller chip.  It should be called once and only
* once for a specified SCSI controller. Since it allocates memory
* for a structure needed by all routines in xxxLib, it must be called before
* any other routines in the library. After calling this routine,
* xxxCtrlInit() should be called at least once before any SCSI transactions
* are initiated using the SCSI controller.
*
* RETURNS: A pointer to XXX_SCSI_CTRL structure, or NULL if memory
* is unavailable or there are invalid parameters.
*/
XXX_SCSI_CTRL *xxxCtrlCreate
    (
    UINT8 *baseAdrs,            /* base address of the SCSI controller */
    UINT   clkPeriod,           /* clock controller period (nsec*100) */
    UINT16 devType              /* XXX SCSI device type */
    )

    {
    FAST SIOP *pSiop;           /* ptr to SCSI controller info */

    /* check that dma buffers are cache-coherent */

    /* cacheDmaMalloc the controller structure and other driver structures */

    pScsiCtrl = (SCSI_CTRL *) pSiop;

    /* inform the SCSI libraries about the size of an XXX event and thread */

    pScsiCtrl->eventSize  = sizeof (XXX_EVENT);
    pScsiCtrl->threadSize = sizeof (XXX_THREAD);

    pScsiCtrl->scsiTransact          = (FUNCPTR)      scsiTransact;
    pScsiCtrl->scsiEventProc           = (VOIDFUNCPTR) xxxEvent;
    pScsiCtrl->scsiThreadInit          = (FUNCPTR)     xxxThreadInit;
    pScsiCtrl->scsiThreadActivate      = (FUNCPTR)     xxxThreadActivate;
    pScsiCtrl->scsiThreadAbort         = (FUNCPTR)     xxxThreadAbort;
    pScsiCtrl->scsiBusControl          = (FUNCPTR)     xxxScsiBusControl;
    pScsiCtrl->scsiXferParamsQuery     = (FUNCPTR)     xxxXferParamsQuery;
    pScsiCtrl->scsiXferParamsSet       = (FUNCPTR)     xxxXferParamsSet;
    pScsiCtrl->scsiWideXferParamsQuery = (FUNCPTR)    xxxWideXferParamsQuery;
    pScsiCtrl->scsiWideXferParamsSet   = (FUNCPTR)     xxxWideXferParamsSet;
```

```
    /* the following virtual functions are not used with this driver */

    pScsiCtrl->scsiDevSelect = NULL;
    pScsiCtrl->scsiInfoXfer  = NULL;

    /* fill in generic SCSI info for this controller */

    scsiCtrlInit (&pSiop->scsiCtrl);

    /* fill in SCSI controller  specific data for this controller */

    /* initialize controller state variables  */

    /*
     * Initialize fixed fields in client shared data area. This "shared"
     * area of memory is shared between this driver and the scripts I/O
     * processor. Fields like data pointers, data size, message pointer,
     * message size, status pointer and size, etc. are typically the
     * pieces of information shared. These fields are updated and managed
     * before and after an I/O process.
     */

    xxxSharedMemInit (pSiop, pSiop->pClientShMem);

    /* spawn SCSI manager - use generic code from "scsiLib.c" */

    pScsiCtrl->scsiMgrId = taskSpawn (xxxScsiTaskName,
                                      xxxScsiTaskPriority,
                                      xxxScsiTaskOptions,
                                      xxxScsiTaskStackSize,
                                      (FUNCPTR) scsiMgr,
                                      (int) pSiop, 0, 0, 0, 0, 0, 0, 0, 0,0);

    return (pSiop);
    }

/*******************************************************************************
*
* xxxCtrlInit - initialize a XXX SCSI controller structure
*
* This routine initializes an  SCSI controller structure, after the structure
* is created with xxxCtrlCreate().  This structure must be initialized before
* the SCSI controller can be used.  It may be called more than once if
* needed;however,it should only be called while there is no activity on the
* SCSI interface. A detailed description of the input parameters follows:
*
* RETURNS: OK, or ERROR if parameters are out of range.
*/

STATUS xxxCtrlInit
    (
    FAST XXX_SCSI_CTRL *pSiop,      /* ptr to SCSI controller struct */
    int scsiCtrlBusId               /* SCSI bus ID of this SCSI controller */
    )
```

```
    {
    SCSI_CTRL * pScsiCtrl = (SCSI_CTRL *) pSiop;

    /* initialize the SCSI controller */

    xxxHwInit (pSiop);

    /*
     * Put the scripts I/O processor in a state whereby it is ready for
     * selections or reselection from the SCSI bus. Such a state continues
     * until either a selection or selection occurs or the driver interrupts
     * the scripts processor and resets its program counter to begin
     * execution elsewhere.
     */

    xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread,
                    XXX_SCRIPT_WAIT);

    return (OK);
    }

/*******************************************************************************
*
* xxxHwInit - initialize the  SCSI controller chip to a known state
*
* RETURNS: N/A
*/

LOCAL void xxxHwInit
    (
    FAST SIOP *pSiop       /* ptr to a SCSI controller info structure */
    )
    {
    /* initialize hardware independent registers */
    }

/*******************************************************************************
*
* xxxScsiBusReset - assert the RST line on the SCSI bus
*
* Issue a SCSI Bus Reset command to the XXX SCSI controller. This should put
* all devices on the SCSI bus in an initial quiescent state.
*
* RETURNS: N/A
*/

LOCAL void xxxScsiBusReset
    (
    FAST SIOP *pSiop    /* ptr to SCSI controller info */
    )
    {
    /* set appropriate register values in order to reset the SCSI bus */
    }
```

```
/*******************************************************************************
*
* xxxIntr - interrupt service routine for the SCSI controller
*
* Find the event type corresponding to this interrupt, and carry out any
* actions which must be done before the SCSI controller is re-started.
* Determine  whether or not the SCSI controller is connected to the bus
* (depending on the event type - see note below).  If not, start a client
* script if possible or else just make the SCSI controller wait for something
* else to happen.
*
* Notify the SCSI manager of a controller event.
*
* RETURNS: N/A
*/

void xxxIntr
    (
    SIOP *pSiop
    )
    {
    XXX_EVENT    event;
    SCSI_EVENT   pScsiEvent = (SCSI_EVENT *) &event;

    BOOL connected = FALSE;
    BOOL notify    = TRUE;
    int  oldState  = (int) pSiop->state;

    /* Save (partial) SCSI controller register context in current thread */

    /* Get event type */

    pScsiEvent-type = xxxEventTypeGet (pSiop);

    /* fill in event information based upon the nature of the event */

    /* controller is now idle: if possible, make it run a script. */

    xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread,
                    XXX_SCRIPT_WAIT);

    /* Send the event to the SCSI manager to be processed. */

    scsiMgrEventNotify ((SCSI_CTRL *) pSiop, pScsiEvent, sizeof (event));
    }

/*******************************************************************************
*
* xxxEventTypeGet - parse SCSI and DMA status registers at interrupt time
*
* RETURNS: an interrupt (event) type code
*/
LOCAL int xxxEventTypeGet
    (
    SIOP * pSiop
    )
```

```
    {
    /* Read interrupt status registers */

    key = intLock ();

    /* Check for fatal errors first */

    /* No fatal errors; try the rest (order of tests is important) */

    return (INTERRUPT_TYPE);
    }

/*******************************************************************************
*
* xxxThreadActivate - activate a SCSI connection for an initiator thread
*
* Set whatever thread/controller state variables need to be set.  Ensure that
* all buffers used by the thread are coherent with the contents of the
* system caches (if any).
*
* Set transfer parameters for the thread based on what its target device
* last negotiated.
*
* Update the thread context (including shared memory area) and note that
* there is a new client script to be activated (see "xxxActivate()").
*
* Set the thread's state to ESTABLISHED.
* Do not wait for the script to be activated.  Completion of the script is
* signalled by an event which is handled by "xxxEvent()".
*
* RETURNS: OK or ERROR
*/
LOCAL STATUS xxxThreadActivate
    (
    SIOP *       pSiop,                 /* ptr to controller info */
    XXX_THREAD * pThread                /* ptr to thread info     */
    )
    {
    scsiCacheSynchronize (pScsiThread, SCSI_CACHE_PRE_COMMAND);

    scsiWideXferNegotiate (pScsiCtrl, pScsiTarget, WIDE_XFER_NEW_THREAD);
    scsiSyncXferNegotiate (pScsiCtrl, pScsiTarget, SYNC_XFER_NEW_THREAD);

    if (xxxThreadParamsSet (pThread, pScsiTarget->xferOffset,
                                     pScsiTarget->xferPeriod) != OK)
        return (ERROR);

    /* Update thread context; activate the thread */

    xxxThreadUpdate (pThread);

    if (xxxActivate (pSiop, pThread) != OK)
        return (ERROR);

    pScsiCtrl->pThread = pScsiThread;
```

```
    xxxThreadStateSet (pThread, SCSI_THREAD_ESTABLISHED);

    return (OK);
    }

/*******************************************************************************
*
* xxxThreadAbort - abort a thread
*
* If the thread is not currently connected, do nothing and return FALSE to
* indicate that the SCSI manager should abort the thread.
*
* RETURNS: TRUE if the thread is being aborted by this driver (i.e. it is
* currently active on the controller, else FALSE.
*/
LOCAL BOOL xxxThreadAbort
    (
    SIOP *       pSiop,                 /* ptr to controller info */
    XXX_THREAD * pThread                /* ptr to thread info     */
    )
    {
    xxxAbort (pSiop);
    xxxThreadStateSet (pThread, SCSI_THREAD_ABORTING);

    return (TRUE);
    }

/*******************************************************************************
*
* xxxEvent - XXX SCSI controller event processing routine
*
* Parse the event type and act accordingly.  Controller-level events are
* handled within this function, and the event is then passed to the current
* thread (if any) for thread-level processing.
*
* RETURNS: N/A
*/
LOCAL void xxxEvent
    (
    SIOP *      pSiop,
    XXX_EVENT * pEvent
    )
    {
    SCSI_CTRL  * pScsiCtrl  = (SCSI_CTRL *)  pSiop;
    SCSI_EVENT * pScsiEvent = (SCSI_EVENT *) pEvent;
    XXX_THREAD * pThread    = (XXX_THREAD *) pScsiCtrl->pThread;

    /* Do controller-level event processing */

    /* If there's a thread on the controller, forward the event to it */
    if (pThread != 0)
        xxxThreadEvent (pThread, pEvent);
    }
```

```
/*******************************************************************************
*
* xxxThreadEvent - SCSI controller thread event processing routine
*
* Forward the event to the proper handler for the thread's current role.
*
* If the thread is still active, update the thread context (including
* shared memory area) and resume the thread.
*
* RETURNS: N/A
*/
LOCAL void xxxThreadEvent
    (
    XXX_THREAD * pThread,
    XXX_EVENT *  pEvent
    )
    {
    SCSI_EVENT *  pScsiEvent  = (SCSI_EVENT *)  pEvent;
    SCSI_THREAD * pScsiThread = (SCSI_THREAD *) pThread;
    SIOP *        pSiop       = (SIOP *)        pScsiThread->pScsiCtrl;
    XXX_SCRIPT_ENTRY entryPt;

    switch (pScsiThread->role)
        {
        case SCSI_ROLE_INITIATOR:
            xxxInitEvent (pThread, pEvent);

            entryPt = XXX_SCRIPT_INIT_CONTINUE;
            break;

        case SCSI_ROLE_IDENT_INIT:
            xxxInitIdentEvent (pThread, pEvent);

            entryPt = XXX_SCRIPT_INIT_CONTINUE;
            break;

        case SCSI_ROLE_IDENT_TARG:
            xxxTargIdentEvent (pThread, pEvent);

            entryPt = XXX_SCRIPT_TGT_DISCONNECT;
            break;

        case SCSI_ROLE_TARGET:
        default:
            logMsg ("xxxThreadEvent: thread 0x%08x: invalid role (%d)\n",
                    (int) pThread, pScsiThread->role, 0, 0, 0, 0);

            entryPt = XXX_SCRIPT_TGT_DISCONNECT;
            break;
        }

    /* Resume thread if it is still connected */

    xxxResume (pSiop, pThread, entryPt);
    }
```

```
/*******************************************************************************
*
* xxxResume - resume a script corresponding to a suspended thread
*
* NOTE: the script can only be resumed if the controller is currently idle.
* To avoid races, interrupts must be locked while this is checked and the
* script re-started.
*
* Reasons why the controller might not be idle include SCSI bus reset and
* unexpected disconnection, both of which might occur in practice.  Hence
* this is not considered to be a major software error.
*
* RETURNS: OK, or ERROR if the controller is in an invalid state (this
* should not be treated as a major software failure).
*/

LOCAL STATUS xxxResume
    (
    SIOP *           pSiop,          /* ptr to controller info          */
    XXX_THREAD *     pThread,        /* ptr to thread info              */
    XXX_SCRIPT_ENTRY entryId         /* entry point of script to resume */
    )
    {
    STATUS status;
    int    key;

    /*
     *  Check validity of connection and start script if OK
     */
    key = intLock ();

    xxxScriptStart (pSiop, pThread, entryId);

    pSiop->state = NCR810_STATE_ACTIVE;
    status = OK;

    intUnlock (key);
    return (status);
    }

/*******************************************************************************
*
* xxxInitEvent - XXX SCSI controller initiator thread event processing route
*
* Parse the event type and handle it accordingly.  This may result in state
* changes for the thread, state variables being updated, etc.
*
* RETURNS: N/A
*/
LOCAL void xxxInitEvent
    (
    XXX_THREAD * pThread,
    XXX_EVENT *  pEvent
    )
    {
    }
```

```
/*******************************************************************************
*
* xxxSharedMemInit - initialize the fields in a shared memory area
*
* Initialize pointers and counts for all message transfers.  These are
* always directed to buffers provided by the SCSI_CTRL structure.
*
* RETURNS: N/A
*/
LOCAL void xxxSharedMemInit
    (
    SIOP *       pSiop,
    XXX_SHARED * pShMem
    )
    {
    }

/*******************************************************************************
*
* xxxThreadInit - initialize a client thread structure
*
* Initialize the fixed data for a thread (i.e., independent of the command).
* Called once when a thread structure is first created.
*
* RETURNS: OK, or ERROR if an error occurs
*/

LOCAL STATUS xxxThreadInit
    (
    SIOP *       pSiop,
    XXX_THREAD * pThread
    )
    {
    scsiThreadInit (&pThread->scsiThread);
    return (OK);
    }

/*******************************************************************************
*
* xxxActivate - activate a script corresponding to a new thread
*
* Request activation of (the script for) a new thread, if possible; do not
* wait for the script to complete (or even start) executing.  Activation
* is requested by signaling the controller, which causes an interrupt.
* The script is started by the ISR in response to this event.
*
* NOTE: Interrupt locking is required to ensure that the correct action
* is taken once the controller state has been checked.
*
* RETURNS: OK, or ERROR if the controller is in an invalid state (this
* indicates a major software failure).
*/
LOCAL STATUS xxxActivate
    (
    SIOP *       pSiop,
```

```
    XXX_THREAD * pThread
    )
    {
    key = intLock ();

    /* Activate controller for the current thread */

    intUnlock (key);

    return (status);
    }

/*******************************************************************************
*
* xxxAbort - abort the active script corresponding to the current thread
*
* Check that there is currently an active script running.  If so, set the
* SCSI controller Abort flag which halts the script and causes an
* interrupt.
*
* RETURNS: N/A
*/

LOCAL void xxxAbort
    (
    SIOP * pSiop                        /* ptr to controller info */
    )
    {
    STATUS status;
    int    key;

    key = intLock ();

    /* Abort the active script corresponding to the current thread */

    intUnlock (key);
    }

/*******************************************************************************
*
* xxxScriptStart - start the SCSI controller executing a script
*
* Restore the SCSI controller register context, including the shared memory
* area, from the thread context.  Put the address of the script entry point
* into the DSP register.  If not in single-step mode, start the script.
*
* NOTE: should always be called with SCSI controller's interrupts locked.
*
* RETURNS: N/A
*/

LOCAL void xxxScriptStart
    (
    SIOP            *pSiop,      /* pointer to  SCSI controller info */
    XXX_THREAD      *pThread,    /* ncr thread info */
    XXX_SCRIPT_ENTRY entryId     /* routine address entry point */
```

```
    )
    {
    static ULONG * xxxScriptEntry [] =
        {
        xxxWait,                        /* wait for re-select or host cmd */
        xxxInitStart,                   /* start an initiator thread */
        xxxInitContinue,                /* continue an initiator thread */
        xxxTgtDisconnect,               /* disconnect a target thread */
        };

    /* Restore the SCSI controller register context for this thread. */
    /*
     *  Set the shared data address, load the script start address,
     *  then start the SCSI controller.
     */

    }

/*******************************************************************************
*
* xxxXferParamsQuery - get (synchronous) transfer parameters
*
* Updates the synchronous transfer parameters suggested in the call to match
* the XXX SCSI controller's capabilities.  Transfer period is in SCSI units
* (multiples * of 4 ns).
*
* RETURNS: OK
*/

LOCAL STATUS xxxXferParamsQuery
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to controller info       */
    UINT8     *pOffset,                 /* max REQ/ACK offset  [in/out] */
    UINT8     *pPeriod                  /* min transfer period [in/out] */
    )
    {
    return (OK);
    }

/*******************************************************************************
*
* xxxWideXferParamsQuery - get wide data transfer parameters
*
* Updates the wide data transfer parameters suggested in the call to match
* the XXX SCSI controller's capabilities.  Transfer width is in the units
* of the WIDE DATA TRANSFER message's transfer width exponent field. This is
* an 8 bit field where 0 represents a narrow transfer of 8 bits, 1 represents
* a wide transfer of 16 bits and 2 represents a wide transfer of 32 bits.
*
* RETURNS: OK
*/
```

```
LOCAL STATUS xxxWideXferParamsQuery
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to controller info       */
    UINT8     *xferWidth                /* suggested transfer width     */
    )
    {
    }

/*******************************************************************************
*
* xxxXferParamsSet - set transfer parameters
*
* Validate the requested parameters, convert to the XXX SCSI controller's
* native format and save in the current thread for later use (the chip's
* registers are not actually set until the next script activation for this
* thread).
*
* Transfer period is specified in SCSI units (multiples of 4 ns).  An offset
* of zero specifies asynchronous transfer.
*
* RETURNS: OK if transfer parameters are OK, else ERROR.
*/

LOCAL STATUS xxxXferParamsSet
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to controller info */
    UINT8      offset,                  /* max REQ/ACK offset     */
    UINT8      period                   /* min transfer period    */
    )
    {
    }

/*******************************************************************************
*
* xxxWideXferParamsSet - set wide transfer parameters
*
* Assume valid parameters and set the XXX's thread parameters to the
* appropriate values. The actual registers are not written yet, but will
* be written from the thread values when it is activated.
*
* Transfer width is specified in SCSI transfer width exponent units.
*
* RETURNS: OK
*/

LOCAL STATUS xxxWideXferParamsSet
    (
    SCSI_CTRL *pScsiCtrl,               /* ptr to controller info */
    UINT8      xferWidth                /* wide data transfer width */
    )
    {
    }
```

Example 4-3 **Advanced I/O Processor Driver Example**

```
; xxxInit.n Script I/O processor assembly code for xxxLib Driver
;
; Copyright 1989-1996 Wind River Systems, Inc.
;
;/*
;Modification history
;--------------------
;01a,28jun95,jds    Created. Adapted from ncr710init.n
;
;
;INTERNAL
;This file contains the assembly level SCSI scripts instructions which are
;used in conjunction with a higher level controller driver. To operate in
;SCSI SCRIPTS mode the SCSI I/O Processor requires only a SCRIPTS start
;address and a signal to begin operation. At that point, the processor
;begins fetching instructions from external memory and then executes them.
;The start address is written to the DMA SCRIPTS Pointer (DSP) register,
;which acts like a typical program counter. All SCRIPT instructions are
;fetched from external memory. The SCSI I/O Processor fetches and executes
;its own instructions by becoming a bus master on the host bus. Instructions
;are executed until a SCSI SCRIPTS interrupt instruction is encountered or
;until an unexpected interrupt causes an interrupt to the external
;processor. Once an interrupt is generated, the SCSI I/O Processor halts all
;operations until the interrupt is serviced. The further execution of
;SCRIPTS is then controlled by the SCSI controller driver which decides
;at which entry point should the SCRIPT processor start executing.
;
;There are four SCRIPT entry points which could be used by the controller
;driver. Execution thereafter is a function of the logic flow within the
;SCRIPTS and cannot be controlled by the driver. Thus, control is
;transferred to the SCRIPTS processor by the controller driver at well known
;entry points and this control is returned to the controller driver by the
;SCRIPTS by generating a SCRIPTS interrupt. The four SCRIPTS entry points
;are described below:
;
;1) xxxWait
;   If the SCSI controller is not connected to the bus, this entry point is
;   used. The SCRIPTS processor waits for selection or re-selection by a SCSI
;   target device (which acts as an initiator during selection), or can be
;   interrupted by a new command from the host. This is done by signaling
;   the processor via register bits. Thus this entry point puts the SCRIPTS
;   processor into a passive mode.
;
;2) xxxInitStart
;   This entry point is used to start a new initiator thread or I/O process
;   (in SCSI parlance), selecting a target, sending the identify message and
;   thus establishing the ITL nexus, and then continuing to follow the SCSI
;   protocol as dictated by the SCSI target, which drives the bus; thus,
;   transferring the command, data, messages and status. This processing is
;   actually done, within the code of the xxxInitContinue entry point. i.e
;   if no stopping condition is encountered, execution continues on into the
;   next logical entry point.
;
```

```
;3) xxxInitContinue
;   This entry point resumes a suspended SCSI thread. SCSI threads are
;   when further processing is required by the controller driver and an int
;   instruction is executed. However, when the higher level management has
;   been worked out, control comes back to a suspended thread and the process
;   of cycling through all the SCSI information tranfer phases continues. In
;   essence, this entry point is the "meat" of an I/O process. The following
;   phases are managed by this entry point.
;   DATA_OUT
;   DATA_IN
;   COMMAND
;   STATUS
;   MSG_OUT
;   MSG_IN
;   XXX_ILLEGAL_PHASE
;
;4) xxxTgtDisconnect
;   Disconnects a target from the SCSI bus. It is the last entry point in
;   an I/O process.
;
;The description level of the code is close to assembly language and is
;in fact the language of the SCRIPTS processor. The assembly code is compiled
;using an NCR compiler which generates opcodes in the form of a static C
;language structure, which is then compiled and loaded into memory.
;
;The opcode is a pair of 32-bit words, that allow operations and offsets for
;the SCRIPTS processor. A detailed discussion can be found in the chip's
;programmer's guide. Some of the important instructions and their formats
;are listed below.
;
;block move instruction.
;  move from <offset> when PHASE_NAME
;  ........
;I/O instructions
;  set target
;  wait DISCONNECT
;  wait RESELECT
;  select from <offset>,@jump
;  .........
;read/write register instructions
;  move REG_NAME to SFBR
;SFBR acts like an accumulator allowing branch instructions based on its
;value
;  .........
;
;control transfer instructions
;  jump <Label>
;  int <value> when PHASE_NAME
;  ...........
;
;INTERRUPT SOURCES
;The SCSI I/O Processor has three main kind of interrupt, scsi, dma interrupt
;and script  interrupt. The int instruction allows the controller driver to
;be interrupted with an interrupt value which is stored in the DSPS register.
;*/
```

```
#define  NCR_COMPILE
#include "xxxScript.h"

;/*****************************************************************************
;*
;* xxxWait - wait for re-selection by target, selection by initiator, or
;*           new command from host
;*/

PROC xxxWait:

;setup instructions here

wait   reselect REL(checkNewCmd)

;
; have been re-selected by a SCSI target
;
reselected:

; handle reselects, insert the reselect logic

int     XXX_RESELECTED                ; all seems OK so far

;
; May have a new host command to handle
;
checkNewCmd:

; insert logic for checking if the processor is connected to the bus

int     XXX_READY                     ; processor is ready for a new thread

;/*****************************************************************************
;*
;* xxxInitStart - start new initiator thread, selecting target and
;* continuing to transfer command, data, messages as requested.
;*
;* At this point the script requires some data in the scratch registers.
;* This is the threads context information.
;*
;* When the script finishes, these registers are updated with the new context
;* information
;*/

PROC xxxInitStart:

; If required to identify, select w. ATN and try to transfer IDENTIFY message
; (if this fails, continue silently).  Otherwise, select without ATN.
;
select  atn from OFFSET_DEVICE, REL(checkNewCmd)

; add code to test various processor states and conditions interrupt driver
; if necessary.

jump     REL(nextPhase)
```

```
;/*****************************************************************************
;*
;* xxxInitContinue - resume an initiator thread
;*
;* At this point the script requires the threads context information in
;* scratch registers
;*
;* When the script finishes, these scratch registers are updated with the
;* the latest context information
;*/

PROC xxxInitContinue:

; some setup code...

nextPhase:

; Normal info transfer request processing
;
phaseSwitch:
jump    REL(doDataOut), when DATA_OUT
jump    REL(doDataIn)   if   DATA_IN
jump    REL(doCommand)  if   COMMAND
jump    REL(doStatus)   if   STATUS
jump    REL(doMsgOut)   if   MSG_OUT
jump    REL(doMsgIn)    if   MSG_IN
int     XXX_ILLEGAL_PHASE

;/*****************************************************************************
;*
;* doDataOut - handle DATA OUT phase
;*/
doDataOut:

;...

jump     REL(nextPhase)

;/*****************************************************************************
;*
;* doDataIn - handle DATA IN phase
;*/
doDataIn:

;...

jump    REL(nextPhase)

;/*****************************************************************************
;*
;* doCommand - handle COMMAND phase
;*/
doCommand:

;...
```

```
jump    REL(nextPhase)

;/*****************************************************************************
;*
;* doStatus - handle STATUS phase
;*/
doStatus:

;...

jump    REL(nextPhase)

;*
;* doMsgOut - handle MSG OUT phase
;*/
doMsgOut:

;...

jump    REL(nextPhase)

;/*****************************************************************************
;*
;* doMsgIn - handle MSG IN phase
;*
;* Note: there is little point in having the '810 parse the message type
;* unless it can save the host some work by doing so;  DISCONNECT and
;* COMMAND COMPLETE are really the only cases in point.  Multi-byte messages
;* are handled specially - see the comments below.
;*/
doMsgIn:

;...

int     XXX_MESSAGE_IN_RECVD                    ; driver handles all others

;
; Have received a DISCONNECT message
;
disconn:

;...

int     XXX_DISCONNECTED

;
; Have received a COMMAND COMPLETE message
;
complete:

;...

int     XXX_CMD_COMPLETE

extended:
```

```
int     XXX_EXT_MESSAGE_SIZE

contExtMsg:

int     XXX_MESSAGE_IN_RECVD                          ; at last !

/*****************************************************************************
* xxxTgtDisconnect - disconnect from SCSI bus
*
*/
PROC xxxTgtDisconnect:

;...

disconnect

int     XXX_DISCONNECTED
```

## 4.3 SCSI BSP Interface

The BSP provides the board information to the driver in its invocations of the initialization routines. The main tasks of the BSP **sysScsiInit( )** routine, which is located in a file named **sysScsi.c** (included from the standard **sysLib.c**), are as follows:

- Address all preliminary board-specific hardware initialization.
- Create a controller driver object by invoking the driver's *xxx***CtrlCreate( )** routine and supplying the board-specific hardware information such as the base address to the SCSI controller registers.
- Connect the SCSI controller's interrupt vector to the driver's interrupt service routine (ISR).
- Perform additional driver initialization by invoking the *xxx***CtrlInit( )** routine and optionally the driver's *xxx***HwInit( )** routine supplying board-specific information such as the SCSI initiator bus ID, and specific hardware register values.
- Supply any DMA routines if an external DMA controller is being used and is not part of the SCSI controller driver.

Any other board-specific configurations to initialize SCSI peripheral devices such as hard disks and tapes or block/sequential devices and file systems must also be accomplished by **sysScsi.c**. Such configuration initialization shall be located in **sysScsiConfig( )**.

The following subsection introduces a template **sysScsiInit( )** routine located in **sysScsi.c.**



Example 4-4 **Template for SCSI Initialization in the BSP (sysScsi.c)**

```
/* sysScsi.c - XXX BSP SCSI-2 initialization for sysLib.c */

/* Copyright 1984-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01a,29nov95,jds  written
*/

/*
Description

This file contains the sysScsiInit() and related routines necessary for
initializing the SCSI subsystem for this BSP.
*/


#ifdef  INCLUDE_SCSI

/* external inclusions */

#include "drv/scsi/xxx.h"
#include "tapeFsLib.h"

/*******************************************************************************
* sysScsiInit - initialize XXX SCSI chip
*
* This routine creates and initializes an SIOP structure, enabling use of the
* on-board SCSI port.  It also connects the proper interrupt service routine
* to the desired vector, and enables the interrupt at the desired level.
*
* RETURNS: OK, or ERROR if the control structure is not created or the
* interrupt service routine cannot be connected to the interrupt.
*/

STATUS sysScsiInit ()
    {

    /* perform preliminary board specific hardware initializations */

    /* Create the SCSI controller */
```

```
    if ((pSysScsiCtrl = (SCSI_CTRL *) xxxCtrlCreate
                                        (
                                        (UINT8 *) SCSI_BASE_ADRS,
                                        (UINT)    XXX_40MHZ,
                                                  devType
                                        )) == NULL)
        {
        return (ERROR);
        }

    /* connect the SCSI controller's interrupt service routine */

    if (intConnect (INUM_TO_IVEC (SCSI_INT_VEC),
                    xxxIntr, (int) pSysScsiCtrl) == ERROR)
        {
        return (ERROR);
        }

    /* Enable SCSI interrupts */

    intEnable (SCSI_INT_LVL);

    /* initialize SCSI controller with default parameters (user tuneable) */

    if (xxxCtrlInit ((XXX_SCSI_CTRL *)pSysScsiCtrl,
                         SCSI_DEF_CTRL_BUS_ID) == ERROR)
        return (ERROR);

#if (USER_D_CACHE_MODE &  CACHE_SNOOP_ENABLE)

    scsiCacheSnoopEnable ((SCSI_CTRL *) pSysScsiCtrl);

#else

    scsiCacheSnoopDisable ((SCSI_CTRL *) pSysScsiCtrl);

#endif

    /* Set the appropriate board specific hardware registers for the SIOP */

    if (xxxSetHwRegister ((XXX_SCSI_CTRL *)pSysScsiCtrl, &hwRegs)
        == ERROR)
        return(ERROR);

    /* Include tape support if configured in config.h */

#ifdef  INCLUDE_TAPEFS
    tapeFsInit ();                      /* initialize tapeFs */
#endif  /* INCLUDE_TAPEFS */

    return (OK);

    }
```

## 4.4 The SCSI Driver Development Process

This following are useful tips on how to develop a new SCSI controller. Breaking the project up into small easily managed steps is generally the best approach.



1. Understand the template drivers and the interfaces with the SCSI libraries.
2. Copy the template driver into your new driver directory. Replace the variable routine and macro names with your chosen driver name (for example, *xxx***Show( )** might become **myDriverShow( )**).
3. Make sure that the interrupt mechanism is working correctly so that upon getting a SCSI interrupt, the driver's ISR is invoked. A good method to ensure that the ISR is invoked is to write to a well known location in memory or NVRAM so that upon re-initialization of the board the developer can tell that the ISR was entered. Getting the ISR to work is a major milestone.
4. Get the driver to select a SCSI peripheral device. A SCSI bus analyzer can clarify what is really happening on the bus, and a *xxx***Show( )** routine is also extremely helpful. Selecting a device is the next major milestone.
5. Refine the driver using a standard programming step-wise process until the desired result is achieved.
6. Run the standard Wind River SCSI tests in order to test various aspects of the SCSI bus, including multiple threads, multiple initiators, and multiple peripheral devices working concurrently as well as the performance and throughput of the driver.

## 4.5 Common SCSI Driver Development Issues

This sections discusses common issues and concerns encountered during SCSI driver development.

### 4.5.1 Troubleshooting and Debugging

This section provides several suggestions for troubleshooting techniques and debugging shortcuts.

#### SCSI Cables and Termination

A poor cable connection or poor SCSI termination is one of the most common sources of erratic behavior, of the VxWorks target hanging during SCSI execution, and even of unknown interrupts. The SCSI bus must be terminated at both ends, but make sure that no device in the middle of the daisy chain has pull-up terminator resistors or some other form of termination.

#### SCSI Library Configuration

Check to see that the test does not exceed the memory constraints within the library, such as the permitted number of SCSI threads, the size of the ring buffers, and the stack size of the SCSI manager. In most cases, the default values are appropriate.

#### Data Coherency Problems

Data coherency problems usually occur in hardware environments where the CPU supports data caching. First disable the data caches and verify that data corruption is occurring. If the problem disappears with the caches disabled, then the coherency problem is related to caches. (Caches can usually be turned off in the BSP by **#undef USER_D_CACHE_ENABLE**.) In order to further troubleshoot the data cache coherency problem, use **cacheDmaMalloc( )** in the driver for all memory allocations. However, if hardware snooping is enabled then the problem may lie elsewhere.

#### Data Address in Virtual Memory Environments

If the CPU board has a Memory Management Unit (MMU), then you must be careful when setting data address pointers during Direct Memory Access (DMA) transfers. When DMA is used in this environment, the physical memory address must be used instead of the virtual memory address. This is because during DMA transfers from the SCSI bus, the SCSI or DMA controller is the bus master and therefore the MMU on the CPU cannot translate the virtual address to the physical address. Instead, the macro **CACHE_DMA_VIRT_TO_PHYS** must be used when providing the data address to the DMA controller.

### 4.5.2 Test Suites

The following sections list and describe the tests provided by Wind River. The source code for these test routines is located in the directory *installDir*/**vxworks-6.***x*/**target/src/test/scsi**.

### scsiDiskThruputTest( )

This test partitions a 16MB block device into blocks of sizes 4,096, 65,536, or 1,048,576 bytes. Sectors consist of blocks of 512 bytes. This test writes and reads the block size to the disk drive and calculates the time taken, thus computing the throughput.

Invoke this test as follows:

```
scsiDiskThruputTest "scsiBusId devLun numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

*scsBusId*
Target device ID

*devLun*
Device logical unit ID

*numBlocks*
Number of blocks in block device

*blkOffset*
Address of first block in volume

For example:

```
scsiDiskThruputTest "4 0 0x0000 0x0000"
```

### scsiDiskTest( )

This test performs any or all of the tests described below. The invocation for **scsiDiskTest( )** is as follows:

```
scsiDiskTest "test scsiBusId devLun Iterations numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

*test*
One of the following:

#1: runs only **commonCmdsTest( )**
#2: runs only **directRwTest( )**
#3: runs only **directCmdsTest( )**
-[a]: runs all disk tests

*scsBusId*
Target device ID

*devLun*
Device logical unit ID

*Iterations*
Number of times to execute read/write tests

*numBlocks*
Number of blocks in block device

*blkOffset*
Address of first block in volume

For example, the following invocation exercises all disk tests, repeating the read/write exercise 10 times:

```
scsiDiskTest "-a 4 0 10 0x0000 0x0000"
```

The default test mode is to execute all of the following three tests.

**commonCmdsTest( )**
This test exercises all mandatory SCSI common-access commands for SCSI peripheral devices. These common access commands are:

- **TEST UNIT READY**
- **REQUEST SENSE**
- **INQUIRY**

**directRwTest( )**
This test exercises write, read, and check data pattern for:

- 6-byte SCSI commands
- 10-byte SCSI commands

**directCmdsTest( )**
This test exercises all of the direct-access commands listed below. Optionally, the **FORMAT** command can be tested by specifying a value of **TRUE** for the parameter *doFormat*.

- **MODE SENSE**
- **MODE SELECT**
- **RESERVE**
- **RELEASE**
- **READ CAPACITY**
- **READ**
- **WRITE**
- **START STOP UNIT**
- **FORMAT** (optional)

### scsiSpeedTest( )

This test initializes a block device for use with a dosFs file system. The test uses a large buffer to read and write from and to contiguous files with both buffered and non-buffered I/O.

**scsiSpeedTest( )** runs a number of laps, and uses timex to time the write and read operations. The speed test should be run on only one drive at a time to obtain maximum throughput.

Invoke this test as follows:

```
scsiSpeedTest "scsiBusId devLun numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

*scsBusId*
: Target device ID

*devLun*
: Device logical unit ID

*numBlocks*
: Number of blocks in block device

*blkOffset*
: Address of first block in volume

For example:

```
scsiSpeedTest "4 0 0x0000 0x0000"
```

### tapeFsTest( )

This test creates a tape file system and issues various commands to test the tape device. You can choose to test fixed-block-size tape devices, variable-block-size tape devices, or both. Fixed-block tests assume 512-byte blocks.

The invocation for **tapeFsTest( )** is as follows:

```
tapeFsTest "test scsiBusId devLun"
```

The individual parameters must fit the guidelines described below:

*test*
One of the following:

**-f** runs only the fixed-block-size test
**-v** runs only the variable-block-size test
**-a** runs both tests

*scsBusId*
Target device ID

*devLun*
Device logical unit ID

For example, the following invocation exercises both tests:

```
tapeFsTest "-a 1 0"
```

# 5 Timestamp Drivers


5.1 Introduction 141

5.2 Timestamp Driver Overview 142

5.3 Timestamp Driver Configuration and BSP Interface 163

5.4 The Timestamp Driver Development Process 166

5.5 Common Timestamp Driver Development Issues 170


## 5.1 Introduction

**NOTE:** The information in the chapter is provided for reference purposes only. You should use this information to maintain existing timer driver code. If you want to develop a new timer driver, see *VxWorks Device Driver Developers Guide, Volume 1* and *VxWorks Device Driver Developer's Guide (Vol. 2): Timer Drivers*.

Detailed monitoring of real-time application performance requires timing information based on high-resolution timers. You can extend the range of information available from VxWorks kernel instrumentation by supplying a *timestamp driver*. For example, if a timestamp driver is available, a precise chronology can be displayed by the Wind River System Viewer, a graphical analysis tool for real-time and embedded systems based on VxWorks.

The timer is a hardware facility; a timestamp driver is a software interface to that facility. This document describes the standard interfaces for a VxWorks timestamp driver, and discusses the requirements for a hardware timer to be used with VxWorks kernel instrumentation. It is not a step-by-step tutorial on the process of writing a timestamp driver.

This chapter is meant for the following readers:

- VxWorks users who need to add a timestamp driver to an existing BSP.
- VxWorks users who wish to use an existing VxWorks timestamp driver in their own applications.

This chapter assumes that the reader has a working knowledge of the target board hardware. No knowledge of the VxWorks kernel or of the System Viewer is assumed, although experience writing device drivers is helpful.

## 5.2 Timestamp Driver Overview

This section provides an overview of the timestamp driver environment. It includes information on hardware characteristics as well as information on the VxWorks interface.

### 5.2.1 Hardware Environment

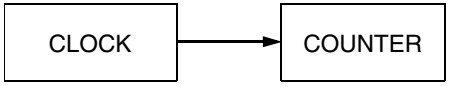
This section discusses typical hardware timer modes of operation and characteristics. This section also defines the VxWorks requirements for timestamp drivers.

#### Modes of Operation

Most target boards have multiple hardware timers available for operating system and application use. The characteristics of timers vary widely due to evolving hardware technology. However, many different types of timers are suitable for use with VxWorks.

In its most basic form, a timer is simply a timing source (that is, a clock) used as input to a counter. The counter counts up or down as the associated clock transitions.

Figure 5-1 **Basic Form of Timer**




There are three common modes in which timers operate: *periodic*, *one-shot*, and *timestamp*. Many newer timers are versatile and can be used in any one of these modes, depending on how they are configured. The characteristics of each mode are as follows:

Periodic Interrupt Timer

The timer counts up or down to a programmed value (called the terminal count), at which point it generates a hardware interrupt. The counter is reset (either by hardware or software), and begins to count up or down again towards the terminal count. The interrupt is the sole output of a periodic interrupt timer. After acknowledging the interrupt, an interrupt service routine (ISR) usually calls an operating system facility to log the interrupt as a *clock tick*. In some cases, the ISR calls an application-specific routine instead.

The terminal count may be adjusted so that an interrupt is generated at a specified time interval. For example, if the terminal count is set such that an interrupt occurs every 10 msec, 100 ticks per second are generated (100Hz).

The VxWorks system and auxiliary clocks use the underlying hardware timers in periodic interrupt mode.

One-Shot Timer

The timer counts up or down to a programmed terminal count, at which point it generates a hardware interrupt. The counter is then disabled (either by hardware or software). An ISR acknowledges the interrupt, and then calls a user-specified routine.

Currently, VxWorks does not support a one-shot timer facility in hardware, although this type of timer can be simulated by having a periodic interrupt timer disable the counter in the ISR. One-shot functionality is provided by the watchdog software module.

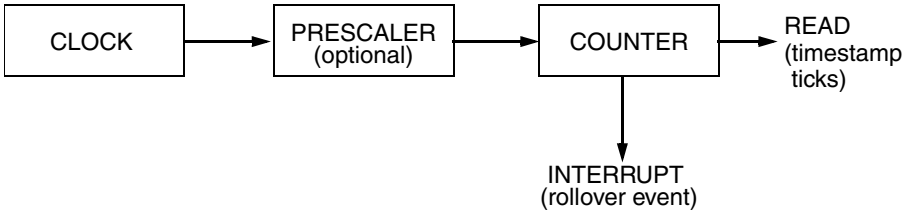
Timestamp Timer

The timer counts up or down to its maximum count (typically, 0 or **MAX_INIT**) at which point it generates a hardware interrupt. The counter rolls over and begins to count again towards the maximum value. After acknowledging the interrupt, an ISR calls an operating system facility or application-specific routine to log the counter rollover. At any time, the operating system or

application may read the counter value to obtain high-resolution timing information in *timestamp tick* units.

This mode of operation differs from a periodic interrupt timer in that the counter is usually allowed to count to its maximum value. Additionally, the counter value is the primary output of the timestamp timer, and the interrupt is only used to announce a counter rollover. Timestamp timer components are typically similar to Figure 5-2.

Figure 5-2 **Components of a Timestamp Timer**




The remainder of this chapter deals only with timers operating in timestamp mode.

### Characteristics of Hardware Timers

Several factors determine how suitable a particular hardware timer may be for a timestamp driver. This information may help you to choose an appropriate timer, if several are available.

#### Read While Enabled

The most important characteristic of a good timestamp timer is the ability to read the counter's value without having to stop the timer from counting. If the timer must be disabled to read the timestamp value accurately, the time spent without the timer running is not recorded, although the system is actually doing work and other timers are continuing to run (the system clock, for instance). This situation is commonly called *time skew*. As time skew accumulates, the timestamp values become more and more removed from the absolute time of the system, as kept by the system clock. Additionally, interrupts must be locked out while the timer is stopped. Both of these effects are detrimental to real-time systems.

#### Prescaler Counter

The input clock is often passed through a prescaler counter to divide the input clock frequency, thereby producing a lower frequency input for the timestamp counter. Although a prescaler is not always present, it can be a useful way of tuning timer devices that have an unusually high input clock frequency. Using a timer frequency significantly greater than your application demands can hamper real-time performance by increasing the number of cycles spent servicing the timer interrupt.

#### Counter Width

The timer's counter should be at least 16 bits wide, although a 24- or 32-bit counter is preferable. The wider a counter, the less often it must roll over, and therefore the less system overhead its ISR incurs. The input frequency can also be higher with a wide counter, which yields more accurate timing information.

#### Preload After Disable

Some timers require that the counter be preloaded with a value before counting resumes. This is an issue only for timers that cannot be read while enabled. This characteristic adds to the time spent with the timer disabled, thereby increasing time-skew problems. The preload mechanism itself provides a way to correct skew, but determining the amount of the correction is difficult; see the discussion of counter preloading in *5.4.2 Working Around Deficiencies In Hardware Timers*, p.167.

#### Cache Coherency

As with all hardware devices, the locations of timer device registers must be cache coherent. This ensures that reads and writes to timer registers are actually accessing the register locations themselves, and not CPU data cache locations. If data cache memory exists, and there is no hardware mechanism (such as an MMU) to guarantee data cache coherency for register locations, the timestamp timer driver must make explicit calls to flush and invalidate the CPU's data cache. This adds to the overhead of reading the timestamp tick value.

### VxWorks Requirements for Timestamp Timers

The VxWorks kernel instrumentation uses a timestamp timer, when available, to log timing information for selected operating system events—for example, semaphore gives and takes, task spawns and deletions, system clock ticks, and interrupts.

VxWorks requires that timestamp timers provide the following features:

Rollover Interrupt
: The timestamp timer must be able to generate a hardware interrupt once the maximum (or terminal) count is reached. An interrupt is needed to avoid aliasing, by announcing the rollover event. Without the interrupt, timestamps are ambiguous, since there is no way to distinguish two timestamps separated by the timer's terminal period.

Fine Resolution
: The timestamp tick *resolution* is calculated as follows:

$$resolution = \frac{1}{timestamp\ tick\ frequency} = \frac{prescaler}{iuput\ clock\ frequency}$$

To be effective, the resolution should be 10 µsec or less (that is, a timestamp tick frequency of at least 100 kHz). Although this is not a strict requirement, it is consistent with timing limitations within the VxWorks kernel. If the timestamp timer output is slower than 100 kHz, some instrumented kernel events may not have distinguishable timing information.

Sizable Period
: The time between timestamp rollovers is the timestamp timer's *period*. The period is defined as the product of the timer resolution and the timer's maximum count:

$$period = (maximum\ count) \times resolution$$

To be effective, the period should be at least 10 msec. If rollovers are more frequent, the overhead of servicing the rollover interrupt may be too intrusive. The greater the period, the better.

### 5.2.2 VxWorks OS Interface

This section discusses how your timestamp driver should interface with the VxWorks operating system.

### Working with the Wind River System Viewer

Although the timestamp timer is meant to be a general facility, some specific information is needed to use it with the kernel instrumentation support for the System Viewer. This section describes the configuration and attachment of the timestamp driver to the VxWorks kernel instrumentation.



#### Attaching the Timestamp Driver to VxWorks

Define **INCLUDE_TIMESTAMP** in *installDir*/**vxworks-6.***x*/**target/config/***bspname*/**config.h** to make the timestamp timer routines available to instrumentation logging routines with **wvTmrRegister( )**.[1] This enables the code in **usrRoot( )** (in *installDir*/**vxworks-6.***x*/**target/config/all/usrConfig.c**) that connects the timestamp driver to the VxWorks kernel instrumentation package.

If you use the standard routine names (described in *5.3 Timestamp Driver Configuration and BSP Interface*, p.163), no other changes are necessary. However, you can also create routines with custom names. This is necessary if a VxWorks timestamp driver is already available for a particular target board, and an alternate driver is to be connected. If this is the case, define **INCLUDE_USER_TIMESTAMP** as well as **INCLUDE_TIMESTAMP** (place the definition in *installDir*/**vxworks-6.***x*/**target/config/***bspname*/**config.h**), to connect the routines named by the **USER_TIME***xxx* macros instead of the default timestamp routines. This does not change the functionality required for any of the routines. It merely provides the ability to connect routines with different names. The connected routines must still adhere to the requirements and functionality specified in *5.3 Timestamp Driver Configuration and BSP Interface*, p.163.

#### Using the System Clock

The kernel instrumentation expects each rollover event to trigger a call to the timestamp callback routine (saved in the variable **sysTimestampRoutine( )**. As described in section *5.4.3 Using the VxWorks System Clock Timer*, p.169, the timestamp driver may use the VxWorks system clock facility. If **sysTimestampConnect( )** returns **ERROR**, the VxWorks kernel instrumentation assumes the system clock is used, and relies on the system clock tick to signal a timestamp timer rollover event.

1. For more information, see the **wvTmrRegister( )** reference entry.

### Timestamp Driver Components

The component concept has been applied to all timer drivers. Driver components are added to domain and bootable application projects in the same way as any other software component.

The generic **TIMESTAMP** component is used to describe and define the common API for all timestamp drivers. However, it does not actually add the code to the build system. One timer driver with timestamp capabilities should be added to the system build in order to provide the timestamp API entry points. Consult the documentation on the particular driver to make sure that it provides timestamp support. Some timer drivers do not provide timestamp support.

### Sample Drivers

The following sections contain skeleton code for three different types of timestamp driver:

- for a hardware timer that *can* be read while enabled
- for a hardware timer that *cannot* be read while enabled
- for systems that have no suitable spare timers, thus requiring that timestamps be derived from the VxWorks system clock timer

For a description of each of these driver types, see *5.4 The Timestamp Driver Development Process*, p.166. For a template driver that you can use as the basis of your own timestamp driver, see *installDir*/**vxworks-6.***x*/**target/src/drv/templateTimer.c**.

Example 5-1 **Timestamp Drivers for Timers that Can Be Read while Enabled**

This example presents a skeleton timestamp device driver for a hardware timer that can be read while enabled. This type of timer is the simplest to configure for timestamp mode. See *5.4.1 Timers that Can Be Read While Enabled*, p.166, for a discussion of the most important details involved in writing this kind of driver.

```
/* sampleATimer.c - sample A timer library */

/* Copyright 1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01a,23mar94,dzb  written.
*/
```

```
/*
DESCRIPTION
This library contains sample routines to manipulate the timer functions on
the sample A chip with a board-independent interface.  This library handles
the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be
defined.

NOTE:  This module provides an example of a VxWorks timestamp timer driver
for a timer that can be read while enabled.  It illustrates the structures
and routines discussed in the documentation "Creating a VxWorks Timestamp
Driver."  This module is only a template.  In its current form,
it does not compile.
*/

#ifdef  INCLUDE_TIMESTAMP

#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleATimer.h"


/* Locals */

LOCAL BOOL    sysTimestampRunning = FALSE;      /* running flag */
LOCAL FUNCPTR sysTimestampRoutine = NULL;       /* user rollover routine */
LOCAL int     sysTimestampArg     = NULL;       /* arg to user routine */

/*******************************************************************************
*
* sysTimestampInt - timestamp timer interrupt handler
*
* This routine handles the timestamp timer interrupt.  A user routine is
* called, if one was connected by sysTimestampConnect().
*
* RETURNS: N/A
*
* SEE ALSO: sysTimestampConnect()
*/

LOCAL void sysTimestampInt (void)
    {
    /* acknowledge the timer rollover interrupt here */

    if (sysTimestampRoutine != NULL)    /* call user-connected routine */
        (*sysTimestampRoutine) (sysTimestampArg);
    }

/*******************************************************************************
*
* sysTimestampConnect - connect a user rtn to the timestamp timer interrupt
*
* This routine specifies the user interrupt routine to be called at each
* timestamp timer interrupt.  It does not enable the timestamp timer itself.
*
```

```
* RETURNS: OK, or ERROR if sysTimestampInt() interrupt handler is not used.
*/

STATUS sysTimestampConnect
    (
    FUNCPTR routine,    /* routine called at each timestamp timer interrupt */
    int arg             /* argument with which to call routine */
    )
    {
    sysTimestampRoutine = routine;
    sysTimestampArg = arg;
    return (OK);
    }
/*******************************************************************************
*
* sysTimestampEnable - initialize and enable the timestamp timer
*
* This routine connects the timestamp timer interrupt and initializes the
* counter registers.  If the timestamp timer is already running, this routine
* merely resets the timer counter. \
*
* Set the rate of the timestamp timer input clock explicitly within the
* BSP, in the sysHwInit() routine.  This routine does not initialize
* the timer clock rate.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be enabled.
*/

STATUS sysTimestampEnable (void)
    {
    if (sysTimestampRunning)
        {
        /* clear the timer counter here */

        return (OK);
        }

    /* connect interrupt handler for the timestamp timer */

    (void) intConnect (INUM_TO_IVEC (XXX), sysTimestampInt, NULL);

    sysTimestampRunning = TRUE;

    /* set the timestamp timer's interrupt vector to XXX (if necessary) */
    /* reset & enable the timestamp timer interrupt */

    /* set the period of timestamp timer (see sysTimestampPeriod()) */

    /* clear the timer counter here */
    /* enable the timestamp timer here */

    return (OK);
    }
```

```
/*******************************************************************************
*
* sysTimestampDisable - disable the timestamp timer
*
* This routine disables the timestamp timer.  Interrupts are not disabled.
* However, the tick counter will not increment after the timestamp timer
* is disabled, ensuring that interrupts are no longer generated.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be disabled.
*/

STATUS sysTimestampDisable (void)
    {
    if (sysTimestampRunning)
        {
        /* disable the timestamp timer here */

        sysTimestampRunning = FALSE;
        }

    return (OK);
    }

/*******************************************************************************
*
* sysTimestampPeriod - get the timestamp timer period
*
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
*
* RETURNS: The period of the timer in timestamp ticks.
*/

UINT32 sysTimestampPeriod (void)
    {
    /*
     * Return the timestamp timer period here.
     * The highest period (maximum terminal count) should be used so
     * that rollover interrupts are kept to a minimum.
     */
    }

/*******************************************************************************
*
* sysTimestampFreq - get the timestamp timer clock frequency
*
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
*
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/

UINT32 sysTimestampFreq (void)
```

```
    {
    UINT32 timerFreq;

    /*
     * Return the timestamp tick output frequency here.
     * This value can be determined from the following equation:
     *     timerFreq = clock input frequency / prescaler
     *
     * When possible, read the clock input frequency and prescaler values
     * directly from chip registers.
     */

    return (timerFreq);
    }

/*******************************************************************************
*
* sysTimestamp - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* Call this routine with interrupts locked.  If interrupts are
* not already locked, use sysTimestampLock() instead.
*
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestampLock()
*/

UINT32 sysTimestamp (void)
    {
    /* return the timestamp timer tick count here */
    }
/*******************************************************************************
*
* sysTimestampLock - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* This routine locks interrupts for cases in which it is necessary to stop
* the tick counter before reading it, or when two independent counters must
* be read.  If interrupts are already locked, use sysTimestamp() instead.
*
* RETURNS: The current timestamp timer tick count.
*
* SEE ALSO: sysTimestamp()
*/

UINT32 sysTimestampLock (void)
    {
    /*
     * Return the timestamp timer tick count here.
     * Interrupts do *not* need to be locked in this routine if
```

```
     * the counter need not be stopped before reading.
     */
    }

#endif  /* INCLUDE_TIMESTAMP */
```

Example 5-2 **Timestamp Drivers for Deficient Timers**

This example presents a skeleton timestamp device driver for a hardware timer that cannot be read while enabled, requires preloading, and counts down. See *5.4.2 Working Around Deficiencies In Hardware Timers*, p.167, for a discussion of the most important details involved in writing this kind of driver.

```
/* sampleBTimer.c - sample B timer library */

/* Copyright 1984-1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01a,23mar94,dzb  written.
*/

/*
DESCRIPTION
This library contains sample routines to manipulate the timer functions on
the sample B chip with a board-independent interface.  This library handles
the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be
defined.

To support the timestamp timer facility, two timers are used:  a counting
timer, and a correction timer.  The counting timer is used as the timestamp
counter, but must be stopped to be read, thereby introducing time skew.  The
correction timer periodically resets the counting timer in an effort to
alleviate cumulative time skew. In addition, the correction timer interrupt
is used for one other purpose: to alert the user to a counting timer reset
(analogous to a timestamp rollover event).

The TS_CORRECTION_PERIOD macro defines the period of the correction timer,
which translates to the period of the counting timer reset (analogous to a
timestamp rollover event).  The TS_SKEW macro can be used to compensate for
time skew incurred when the counting timer is stopped in sysTimestamp() and
sysTimestampLock().  The value of TS_SKEW is subtracted from the stopped
timestamp counter in an attempt to make up for "lost" time.  The correct
value to adjust the timestamp counter is not only board-dependent, it is
influenced by CPU speed, cache mode, memory speed, and so on.

NOTE:  This module provides an example of a VxWorks timestamp timer driver
for a timer that cannot be read while enabled, requires preloading, and
counts down.  It illustrates the structures and routines discussed in the
document "Creating a VxWorks Timestamp Driver."  This module
```

```
is only a template.  In its current form, it does not compile.
*/

/* includes */

#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleBTimer.h"

#ifdef  INCLUDE_TIMESTAMP

/* defines */

#ifndef TS_CORRECTION_PERIOD
#define TS_CORRECTION_PERIOD 0xXXX...   /* timestamp skew correction pd. */
#endif  /* TS_CORRECTION_PERIOD */      /* see sysTimestampPeriod() */

#ifndef TS_SKEW
#define TS_SKEW 0                       /* timestamp skew correction time */
#endif  /* TS_SKEW */

/* locals */

LOCAL BOOL    sysTimestampRunning = FALSE;      /* running flag */
LOCAL FUNCPTR sysTimestampRoutine = NULL;       /* user rollover routine */
LOCAL int     sysTimestampArg     = NULL;       /* arg to user routine */

/*************************************************************************
*
* sysTimestampInt - correction timer interrupt handler
*
* This routine handles the correction timer interrupt.  A user routine is
* called, if one was connected by sysTimestampConnect().
*
* RETURNS: N/A
*
* SEE ALSO: sysTimestampConnect()
*/

LOCAL void sysTimestampInt (void)
    {
    /* acknowledge the correction timer interrupt here */

    sysTimestampEnable ();

    if (sysTimestampRoutine != NULL)    /* call user-connected routine */
        (*sysTimestampRoutine) (sysTimestampArg);
    }

/*************************************************************************
*
* sysTimestampConnect - connect a user routine to the timestamp timer
* interrupt
*
* This routine specifies the user interrupt routine to be called at each
* timestamp timer interrupt.  It does not enable the timestamp timer itself.
*
```

```
* RETURNS: OK, or ERROR if sysTimestampInt() interrupt handler is not used.
*/

STATUS sysTimestampConnect
    (
    FUNCPTR routine,    /* routine called at each timestamp timer interrupt */
    int arg             /* argument with which to call routine */
    )
    {
    sysTimestampRoutine = routine;
    sysTimestampArg = arg;

    return (OK);
    }

/*******************************************************************************
*
* sysTimestampEnable - initialize and enable the timestamp timer
*
* This routine connects the timestamp timer interrupt and initializes the
* counter registers.  If the timestamp timer is already running, this routine
* merely resets the timer counter.
*
* Set the rate of the timestamp timer input clock explicitly within the
* BSP, in the sysHwInit() routine.  This routine does not initialize
* the timer clock rate.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be enabled.
*/

STATUS sysTimestampEnable (void)
    {
    int lockKey;

    if (sysTimestampRunning)
        {
        lockKey = intLock ();                   /* LOCK INTERRUPTS */

        /* disable the counting timer here */

        /* preload the reset count here */

        /* enable the counting timer here */

        /* wait for preload to take effect here */

        intUnlock (lockKey);                    /* UNLOCK INTERRUPTS */

        return (OK);
        }

    /* connect interrupt handler for the correction timer */

    (void) intConnect (INUM_TO_IVEC (XXX), sysTimestampInt, NULL);

    /* set the correction timer's interrupt vector to XXX (if necessary) */
```

```
    sysTimestampRunning = TRUE;

    /* set the period of the correction timer (see sysTimestampPeriod()) */
    /* set the period of the  counting timer = reset count */

    /* enable the counting timer here */
    /* enable the correction timer here */

    /* wait for preload to take effect on both timers here */

    return (OK);
    }

/*******************************************************************************
*
* sysTimestampDisable - disable the timestamp timer
*
* This routine disables the timestamp timer.  Interrupts are not disabled.
* However, the tick counter will not decrement after the timestamp timer
* is disabled, ensuring that interrupts are no longer generated.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be disabled.
*/

STATUS sysTimestampDisable (void)
    {
    if (sysTimestampRunning)
        {
        sysTimestampRunning = FALSE;

        /* disable the correction timer here */
        /* disable the counting timer here */
        }

    return (OK);
    }

/*******************************************************************************
*
* sysTimestampPeriod - get the timestamp timer period
*
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
*
* RETURNS: The period of the timer in timestamp ticks.
*/

UINT32 sysTimestampPeriod (void)
    {
    /*
     * Return the correction timer period here.
     * A reasonable correction period should be chosen.  A short period
     * causes increased CPU overhead due to correction timer interrupts.
```

```
     * A long period allows for a large accumulation of time skew
     * due to sysTimestamp() calls stopping the counting timer.
     */

    return (TS_CORRECTION_PERIOD);
    }

/*******************************************************************************
*
* sysTimestampFreq - get the timestamp timer clock frequency
*
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
*
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/

UINT32 sysTimestampFreq (void)
    {
    UINT32 timerFreq;

    /*
     * Return the timestamp tick output frequency here.
     * This value can be determined from the following equation:
     *     timerFreq = clock input frequency / prescaler
     *
     * When possible, read the clock input frequency and prescaler values
     * directly from chip registers.
     */

    return (timerFreq);
    }

/*******************************************************************************
*
* sysTimestamp - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* Call this routine with interrupts locked.  If interrupts are
* not already locked, use sysTimestampLock() instead.
*
* RETURNS: The current timestamp timer tick count.
*
* SEE ALSO: sysTimestampLock()
*/

UINT32 sysTimestamp (void)
    {
    UINT32 tick = 0;
    register UINT32 * pTick;
    register UINT32 * pPreload;
```

```
    if (sysTimestampRunning)
        {
        /* pTick = counter read register location */
        /* pPreload = counter preload register location */

        /* disable counting timer here */

        tick = *pTick;                  /* read counter value */
        *pPreload = tick - TS_SKEW;     /* set preload value
                                         (with time-skew adjustment) */

        /* enable counting timer here */

        tick -= (0xfff...);             /* adjust to incrementing value */
        }

    return (tick);
    }

/*******************************************************************************
*
* sysTimestampLock - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* This routine locks interrupts for cases in which it is necessary to stop
* the tick counter before reading it, or when two independent counters must
* be read.  If interrupts are already locked, use sysTimestamp() instead.
*
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestamp()
*/

UINT32 sysTimestampLock (void)
    {
    UINT32 tick = 0;
    register UINT32 * pTick;
    register UINT32 * pPreload;
    int lockKey;

    if (sysTimestampRunning)
        {
        lockKey = intLock ();           /* LOCK INTERRUPTS */

        /* pTick = counter read register location */
        /* pPreload = counter preload register location */

        /* disable counting timer here */

        tick = *pTick;                  /* read counter value */
        *pPreload = tick - TS_SKEW;     /* set preload value
                                           (with time-skew adjustment) */
```

```
        /* enable counting timer here */

        intUnlock (lockKey);            /* UNLOCK INTERRUPTS */

        tick -= (0xfff...);             /* adjust to incrementing value */
        }

    return (tick);
    }

#endif   /* INCLUDE_TIMESTAMP */
```

Example 5-3 **Timestamp Drivers for the VxWorks System Clock Timer**

This example presents a skeleton timestamp driver for systems that have no suitable spare timers, so that timestamps must be derived from the VxWorks system clock timer. See *5.4.3 Using the VxWorks System Clock Timer*, p.169, for a discussion of the most important details involved in writing this kind of driver.

```
/* sampleCTimer.c - sample C timer library */

/* Copyright 1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01a,23mar94,dzb  written.
*/

/*
DESCRIPTION
This library contains sample routines to manipulate the timer functions on
the sample C chip with a board-independent interface.  This library handles
the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be
defined.

NOTE:  This module provides an example of a VxWorks timestamp timer driver
implemented by reading the system clock timer counter.  It illustrates the
structures and routines discussed in the document "Creating a
VxWorks Timestamp Driver."  This module is only a template.
In its current form, it does not compile.
*/

#ifdef  INCLUDE_TIMESTAMP

#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleCTimer.h"

/* Locals */

LOCAL BOOL    sysTimestampRunning = FALSE;       /* running flag */
```

```
/*******************************************************************************
*
* sysTimestampConnect - connect a user routine to the timestamp timer
* interrupt
*
* This routine specifies the user interrupt routine to be called at each
* timestamp timer interrupt.  It does not enable the timestamp timer itself.
*
* RETURNS: OK, or ERROR if sysTimestampInt() interrupt handler is not used.
*/

STATUS sysTimestampConnect
    (
    FUNCPTR routine,    /* routine called at each timestamp timer interrupt */
    int arg             /* argument with which to call routine */
    )
    {
    /* ERROR indicates that the system clock tick specifies a
     * rollover event */

    return (ERROR);
    }

/*******************************************************************************
*
* sysTimestampEnable - initialize and enable the timestamp timer
*
* This routine connects the timestamp timer interrupt and initializes the
* counter registers.  If the timestamp timer is already running, this routine
* merely resets the timer counter.
*
* Set the rate of the timestamp timer input clock explicitly within the
* BSP, in the sysHwInit() routine.  This routine does not initialize
* the timer clock rate.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be enabled.
*/

STATUS sysTimestampEnable (void)
    {
    if (sysTimestampRunning)
        return (OK);

    sysTimestampRunning = TRUE;

    sysClkEnable ();             /* ensure the system clock is running */

    return (OK);
    }

/*******************************************************************************
*
* sysTimestampDisable - disable the timestamp timer
*
* This routine disables the timestamp timer.  Interrupts are not disabled.
```

```
* However, the tick counter does not increment after the timestamp timer
* is disabled, ensuring that interrupts are no longer generated.
*
* RETURNS: OK, or ERROR if the timestamp timer cannot be disabled.
*/

STATUS sysTimestampDisable (void)
    {
    sysTimestampRunning = FALSE;
    return (ERROR);
    }

/*******************************************************************************
*
* sysTimestampPeriod - get the timestamp timer period
*
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
*
* RETURNS: The period of the timer in timestamp ticks.
*/

UINT32 sysTimestampPeriod (void)
    {
    /* return the system clock period in timestamp ticks */

    return (sysTimestampFreq ()/sysClkRateGet ())
    }

/*******************************************************************************
*
* sysTimestampFreq - get the timestamp timer clock frequency
*
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
*
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/

UINT32 sysTimestampFreq (void)
    {
    UINT32 timerFreq;

    /*
     * Return the timestamp tick output frequency here.
     * This value can be determined from the following equation:
     *     timerFreq = clock input frequency / prescaler
     * When possible, read the clock input frequency and prescaler values
     * directly from chip registers.
     */

    return (timerFreq);
    }
```

```
/*******************************************************************************
*
* sysTimestamp - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* Call this routine with interrupts locked. If interrupts are
* not already locked, use sysTimestampLock() instead.
*
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestampLock()
*/

UINT32 sysTimestamp (void)
    {
    /* return the system clock timer tick count here */
    }
/*******************************************************************************
*
* sysTimestampLock - get the timestamp timer tick count
*
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* This routine locks interrupts for cases in which it is necessary to stop
* the tick counter before reading it, or when two independent counters must
* be read.  If interrupts are already locked, use sysTimestamp() instead.
*
* RETURNS: The current timestamp timer tick count.
*
* SEE ALSO: sysTimestamp()
*/

UINT32 sysTimestampLock (void)
    {
    /*
     * Return the system clock timer tick count here.
     * Interrupts do *not* need to be locked in this routine if
     * the counter does not need to be stopped to be read.
     */
    }

#endif  /* INCLUDE_TIMESTAMP */
```

## 5.3 Timestamp Driver Configuration and BSP Interface

The timestamp timer interface is non-standard; it does not utilize the VxWorks I/O system. Although the interface was developed for use with VxWorks kernel instrumentation, it is also useful as a general BSP facility. The timestamp driver's external interface may change when a more generic, abstracted timer facility is adopted.

The following sections describe each procedure and its external interface. The descriptions apply to a standard timestamp driver. Although the external functionality must remain as described here, procedure content may differ for a particular driver implementation.

**NOTE:** Remember that each routine must return the appropriate value, as described in the following sections. For example, **sysTimestampEnable( )** must return **OK** if successful, or **ERROR** if not successful. (**OK** and **ERROR** are defined in the VxWorks header file *installDir*/**vxworks-6.***x*/**target/h/vxWorks.h**.)

### sysTimestampConnect( )

This routine specifies the *timestamp callback routine*, a routine to be run each time the timestamp counter rolls over. If this facility is available, **sysTimestampConnect( )** must store the function pointer in the global variable **sysTimestampRoutine** and return **OK**, to indicate success.

If the callback cannot be provided, **sysTimestampConnect( )** returns **ERROR** to indicate that no callback routine is connected. In this situation, the VxWorks kernel instrumentation does not use the interrupt handler **sysTimestampInt( )** as part of its timestamp timer implementation, but relies instead on the system clock tick to signal a timestamp reset event (see *5.4.3 Using the VxWorks System Clock Timer*, p.169). To use the timestamp driver in other applications, you must make similar provisions for an **ERROR** result.

The **sysTimestampConnect( )** routine does not enable the timestamp timer itself.

```
STATUS sysTimestampConnect
    (
    FUNCPTR routine,
    int arg
    )
```

The arguments for this routine are the following:

*routine*
Pointer to the routine called at each timer rollover interrupt.

*arg*
Argument for the routine referenced in the *routine* parameter.

The result must be **OK** or **ERROR**.

#### sysTimestampEnable( )

If the timer is not already enabled, this routine performs all the necessary initialization for the timer (for example, connecting the interrupt vector, resetting registers, configuring for timestamp mode, and so on), and then enables the timestamp timer. If the timer is already enabled, this routine simply resets the timer counter value.

```
STATUS sysTimestampEnable (void)
```

This routine takes no arguments.

The result must be **OK** or **ERROR**.

#### sysTimestampDisable( )

This routine disables the timestamp timer. Interrupts are not disabled. However, the tick counter does not count after the timestamp timer is disabled; thus, rollover interrupts are no longer generated.

```
STATUS sysTimestampDisable (void)
```

This routine takes no arguments.

The result must be **OK** or **ERROR**.

#### sysTimestampPeriod( )

This routine returns the period of the timer in timestamp ticks. The period is the number of ticks the timestamp timer counts before rolling over (or resetting) and restarting the counting process.

```
UINT32 sysTimestampPeriod (void)
```

This routine takes no arguments.

The result must be the period of the timer in timestamp ticks.

### sysTimestampFreq( )

This routine returns the output frequency of the timer, in timestamp ticks per second. When possible, the frequency should be derived from actual hardware register values.

If the timer input clock is programmable, do not set its clock rate in **sysTimestampFreq( )**. Setting the timer input clock rate should be part of the initialization performed by **sysHwInit( )** in **sysLib.c**.

```
UINT32 sysTimestampFreq (void)
```

This routine takes no arguments.

The result must be the timestamp timer frequency, in ticks per second.

### sysTimestamp( )

This routine returns the current value of the timestamp counter, when interrupts are already locked. To convert this tick count to seconds, divide by the result of **sysTimestampFreq( )**. The result must increase; that is, the timestamp values must count up. If you are working with a timer that actually counts down, see *5.4.2 Working Around Deficiencies In Hardware Timers*, p.167.

If interrupts are not already locked, call **sysTimestampLock( )** instead.

```
UINT32 sysTimestamp (void)
```

This routine takes no arguments.

The result must be the current tick count of the timestamp timer.

### sysTimestampLock( )

This routine returns the current value of the timestamp counter. To convert the result to seconds, divide the tick count by the result of **sysTimestampFreq( )**. The result must increase monotonically; that is, the timestamp values must count up. If you are working with a timer that actually counts down, see *5.4.2 Working Around Deficiencies In Hardware Timers*, p.167.

This routine locks interrupts for cases in which it is necessary to stop the tick counter in order to read it, or when two independent counters must be read. If interrupts are already locked, call **sysTimestamp( )** instead.

```
UINT32 sysTimestampLock (void)
```

This routine takes no arguments.

The result must be the current tick count of the timestamp timer.

**NOTE:** Because Wind River System Viewer uses the timestamp driver to log system calls and other basic operating system events, the **sysTimestamp( )** and **sysTimestampLock( )** routines must not make calls that generate these events. For a complete discussion of event logging and examples of operating system facilities that generate System Viewer events, see the *Wind River System Viewer User's Guide*.

## 5.4 The Timestamp Driver Development Process

This section discusses the three cases of timestamp device drivers and how each is developed. These descriptions correspond to the sample code provided in *Sample Drivers*, p.148.

### 5.4.1 Timers that Can Be Read While Enabled

Example 5-1 shows a sample device driver for hardware timers that can be read while enabled. This type of timer is the simplest to configure for timestamp mode, and the device driver code is straightforward.

#### Timer Period

The timer should be configured for the highest possible period by setting the terminal count to its maximum value (usually 0xfff… when counting up, and 0 when counting down).

#### Interrupt Level

If programmable, a high-priority interrupt level should be chosen for boards with a low timer period. This ensures that frequent rollover interrupts are serviced without delay, and that the rollover event is registered in a timely manner with the timestamp callback routine (**sysTimestampRoutine( )**).



#### Interrupt Locking

Timers that can be read while enabled do not need to lock interrupts in the **sysTimestampLock( )** routine.

### 5.4.2 Working Around Deficiencies In Hardware Timers

The sample device driver in Example 5-2 illustrates techniques for using a hardware timer that cannot be read while enabled, requires preloading, and counts down. This combination of timer attributes presents several problems for the device driver.

#### Timer Re-Synchronization

If a timestamp timer cannot be read while enabled, a second *correction* timer can compensate: use the correction timer to reset the timestamp timer periodically. In this scenario, the second timer runs as a periodic interrupt timer. On each interrupt it resets the first (*counting*) timer. The counting timer is stopped and read for timestamp values, but never generates an interrupt because it is always reset before reaching its terminal count. However, the correction timer does generate interrupts; because it is not read for timestamp values, it never has time-skew problems. The correction timer ISR resets the counting timer, and then calls the timestamp callback routine (**sysTimestampRoutine**).

This approach clears the time skew that accumulates in the counting timer between resets. Although a discernible time skew may be present towards the end of the timer period, it is flushed by the reset operation.

### Timer Period

Because the counting timer is always reset by the correction timer, the timestamp timer period is really the correction timer period. In the Example 5-2 sample code, this period is set by the **TS_CORRECTION_PERIOD** macro. The value must balance a short period's increased interrupt service rate with a long period's noticeable time skew accumulation.

The chosen period should be based on the amount of time skew that can accumulate, which is related to how often the timestamp facility is called and to the sensitivity of the application using the facility. Wind River's experience is that a correction period of 100 to 150 msec sufficiently satisfies both requirements for most applications.

### Down Counter

The timestamp values must increase. If the timer in use actually counts down, the tick count must be converted to an incrementing value. This is easily done by subtracting the counter value from the reset value (usually 0xfff… for a down counter).

### Counter Preloading

If the counter value must be preloaded before the timer can resume counting, three subroutines must perform this action: **sysTimestamp( )**, **sysTimestampLock( )**, and **sysTimestampEnable( )**. The preload operation adds to the time spent with the timer disabled, exacerbating time-skew problems.

After the **sysTimestampEnable( )** routine enables the counting timer, it may need to delay until the preload value is physically loaded into the counter. This is an issue for timers that synchronize the preloading with a prescaler output transition. If a delay is not inserted, it may be possible for a fast target board to execute the timer preload, return from **sysTimestampEnable( )**, and call **sysTimestamp( )**, which stops the timer and specifies a different preload value. This would nullify the **sysTimestampEnable( )** reset operation.

### Adjustment for Time Skew

Counters that are writeable or that have a preload mechanism can compensate for time skew. While the counter is stopped for a read operation, the counter value or

the preload value may be adjusted by adding (for an up counter) or subtracting (for a down counter) the number of ticks spent with the timer disabled. The Example 5-2 sample code subtracts the **TS_SKEW** macro (0, by default) from the stopped timestamp counter in an attempt to make up for lost time. Note that the adjustment value is not only board-dependent, it is influenced by CPU speed, cache mode, memory speed, and so on. In the default case (**TS_SKEW** = 0), compiler optimization eliminates the **TS_SKEW** adjustment.



#### Counter Read Optimization

Write the **sysTimestamp( )** and **sysTimestampLock( )** routines so that the counter and preload register locations are set before the timer is stopped, in order to reduce the time spent with the counter disabled. This minor change causes a significant reduction in time skew.

### 5.4.3 Using the VxWorks System Clock Timer

Example 5-3 presents a sample device driver that reads the VxWorks system clock timer to obtain the timestamp tick count. This approach is useful if there are no other timers available, and if the system clock timer's counter can be read while enabled.

#### Timer Rollover Interrupt

When the system clock timer is used as the timestamp timer, the usual **sysTimestampInt( )** routine cannot be used to service the timer interrupt. This is because the system clock timer already has an ISR. Thus, the system clock tick can be monitored to provide timestamp rollover information. The **sysTimestampConnect( )** routine always returns **ERROR** because the **sysTimestampRoutine** callback routine is not used.

#### Timer Counter Not Reset

Because the system clock is independent of the timestamp facility, the timestamp driver must not disrupt the system clock in any way. Thus, **sysTimestampEnable( )** does not reset the timer counter for the system clock. This causes inaccurate timestamp values until the first system clock tick ISR resets the

timer counter. For similar reasons, **sysTimestampDisable( )** does not physically disable the system clock.

#### Timer Period

The period of the system clock timer is under the control of the system clock facility, not under the control of the timestamp driver. Thus, the system and the application should not call **sysClkRateSet( )** to change the system clock rate once **sysTimestampPeriod( )** has been called to determine the timestamp timer period.

## 5.5 Common Timestamp Driver Development Issues

This sections discusses common issues and concerns encountered during timestamp driver development.

Expect significant changes to the API for all types of timer drivers in the future. Wind River is in the process of developing a new API with an object-oriented interface. This new API corrects the design problem that exists when each driver module provides exactly the same entry points.

# 6 Additional Drivers


6.1 Introduction 171

6.2 ATAPI Drivers 172

6.3 Interrupt Controller Drivers 172

6.4 Memory Drivers 174

6.5 Multi-Mode (SIO) Serial Drivers 176


## 6.1 Introduction

**NOTE:** The information in the chapter is provided for reference purposes only. If you want to develop a new driver, see *VxWorks Device Driver Developers Guide, Volume 1*.

This chapter covers a variety of drivers for different purposes.

## 6.2 ATAPI Drivers

For most situations, the general purpose ATA/ATAPI driver included with VxWorks (*installDir*/**vxworks-6.***x*/**target/src/drv/hdisk/ataDrv.c**) works without modification. The driver uses configurable data access macros which allow the proper BSP routines to be called when the driver interacts with hardware. **ataDrv.c** is monolithic, meaning that its routines perform functions that would otherwise be done in a generic library, as well as performing the actual interaction with hardware. Writing a new driver for ATAPI at this time would involve either altering **ataDrv.c** or extracting its generic functionality, and is not recommended.

## 6.3 Interrupt Controller Drivers

**NOTE:** If you want to develop a new interrupt controller driver, see *VxWorks Device Driver Developers Guide, Volume 1* and *VxWorks Device Driver Developer's Guide (Vol. 2): Interrupt Controller Drivers*.

For VxWorks 6.*x* and later, interrupt controllers are incorporated into the processor abstraction layer (PAL), guidelines for writing these drivers are not available at the time of this printing. For more information, see the Wind River Online Support Web site.

### BSP Interface

This section describes a common organization for interrupt controller driver usage, along with some guidelines on specific details of what to avoid and what to make sure is incorporated. Because the design of interrupt controllers varies widely, this can only be an approximate guide. For more information, refer to the template interrupt controller driver in *installDir*/**vxworks-6.***x*/**target/src/drv/intrCtl/templateIntrCtl.c** and the interrupt controller driver in your reference BSP.

Typical interrupt controller drivers use two initialization routines. Often, they must provide an interrupt service routine and a connect routine.

Interrupt controllers should be initialized early in **sysHwInit( )**. They must be initialized before any device generates an interrupt. Early in processor initialization, the processor's interrupts are masked, so any interrupts which do occur should not cause problems. Although interrupt controller initialization can occur earlier than this, the best design usually does interrupt controller initialization as the first call from **sysHwInit( )**.

Usually, the architecture specific version of **intConnect( )** is called to connect the interrupt controller interrupt source to the architecture specific processor interrupt system. However, **intConnect( )** requires the memory system to be available in order to allocate memory for a dynamically allocated interrupt stub, which calls the actual Interrupt Service Routine (ISR).



For this reason, the external interrupt controller cannot be connected to the architecture specific processor interrupt system until after **sysHwInit( )** is complete and the root task is running. So the appropriate place to put the **intConnect( )** call is at the beginning of **sysHwInit2( )**. Usually, the interrupt controller must be first in **sysHwInit2( )**, because it must be before other interrupts are connected.

#### Non-Vectored Interrupt Sources

In an ideal world, all interrupt sources provide a vector to use for fast interrupt dispatching. In this case, the hardware provides a vector which is used to dispatch the appropriate ISR without the need to handle the interrupt controller directly at interrupt time. Drivers for interrupt controllers with this property may require nothing more than the **sysHwInit( )** and **sysHwInit2( )** initialization routines mentioned above.

However, vectored interrupt sources may not always be available. The interrupt controller's output pin is connected to some interrupt input pin on some other interrupt controller, possibly directly to a processor interrupt, and possibly on some other interrupt controller elsewhere on the board. Because no vector is available, the architecture specific interrupt system does not know what device generated the interrupt. So the interrupt controller driver must query the controller to see which pin generated the interrupt, and dispatch the appropriate ISR. How to do this depends on which processor architecture is being used. Refer to the source code in the *installDir*/**vxworks-6.***x*/**target/src/arch/***ARCH* directory, the template interrupt controller driver, and any interrupt controller driver in the reference BSP.

## 6.4 Memory Drivers

**NOTE:** The information in the chapter is provided for reference purposes only. If you want to develop a new driver, see *VxWorks Device Driver Developers Guide, Volume 1*.

Memory controllers are not related to any of the normal device driver interfaces to the OS. The memory controller is typically configured by boot code early in the boot process when the processor's initial power-on initialization is performed, and not modified afterward.

Memory controllers are often quite simple. The drivers are typically written in assembly and put in the BSP **romInit.s** file. For example, the assembly source for one PowerPC processor's on-chip memory controller is about 60 lines long, including comments and blank lines, and just 34 assembly instructions when comments and blank lines are removed. Some memory controller initialization sequences are even shorter.

Many times, the assembly source for memory controller initialization is provided by the memory controller vendor. This code can often be used with little or no modification. When additional work is required, it usually takes one of several forms, described in the following sections.

### 6.4.1 Hardware Mismatches

The code provided by the controller vendor may not match the type of memory which is being used in your hardware. In this case, modifications may need to be made to handle bank size, bank count, memory speed and divisors, ECC characteristics, and other aspects of memory configuration. If this is the case, you need to work with the memory controller vendor or memory vendor to determine the appropriate settings. Sometimes, it may be better to re-design the hardware to use memory types that are already supported.

### 6.4.2 Complex Modern Memory Controllers

Memory controllers included on-chip on some modern processors have become complex. In this case, it may be better to write the memory controller in C instead of maintaining hundreds, or even thousands, of lines of assembly source code.

There are a couple of potential problems with this, but solutions may be available.

First, in order to use C, a stack must be available for subroutine call overhead. This means some RAM must be available to contain the stack data before the memory controller is configured. If the chip provides a small bank of static RAM, it can sometimes be configured to be available for use by the memory controller driver. If the chip does not already include any on-chip static RAM or if it is not available for other reasons, it may be possible to include a small bank of on-board static RAM for this purpose.

The second problem with using C source code is related to the make subsystem, source code, and the way the bootable image is created. For the purposes of a memory controller driver, the boot ROM image consists of three modules: **romInit.o**, **bootInit.o**, and an object file containing a RAM resident image which is copied to RAM early in the boot process. A more complete description of the process of creating the boot ROM image or standalone VxWorks image is described in the *VxWorks BSP Developer's Guide*. Also, you can find details by examining the output of the **make bootrom** command.

The memory controller must be linked into the image along with **romInit.o** and **bootInit.o**, before the RAM resident image is loaded into RAM. The RAM resident image cannot use it, since the RAM must have already been initialized before this image is run.

The build system includes a mechanism for including object modules in the RAM-resident image, but it does not include a specific mechanism to include additional object modules in the ROM-resident image.

There is an indirect mechanism to provide object modules which are included in the base bootrom image. Although you cannot include an object module directly, you can include a library in the **LIB_EXTRA** macro in **Makefile**. The full makefile additions might look something like the following, extracted from makefile in the **wrPpmc440gp** BSP:

```
LIB_EXTRA        = romExtras.a

# Additional objects used by romInit

EXTRA_OBJS       = romI2cDrv.o romSdramInit.o

romExtras.a: $(EXTRA_OBJS)
        $(AR) crus $@ $(EXTRA_OBJS)
```

For additional information, download the **wrPpmc440gp** BSP to see exactly what is being done and how this situation is handled.

## 6.5 Multi-Mode (SIO) Serial Drivers

**NOTE:** The information in the chapter is provided for reference purposes only. You should use this information to maintain existing serial driver code. If you want to develop a new serial driver, see *VxWorks Device Driver Developers Guide, Volume 1* and *VxWorks Device Driver Developer's Guide (Vol. 2): Serial Drivers*.

The generic *multi-mode serial* (SIO) drivers are provided in the *installDir*/**vxworks-6.***x*/**target/src/drv/sio** directory. These drivers are called SIO drivers to distinguish them from the older serial drivers that have only a single interrupt mode of operation.

SIO drivers provide an interface for setting hardware options, such as the number of stop bits, data bits, parity, and so on. In addition, these drivers provide an interface for polled communication that can provide external mode debugging (such as ROM-monitor style debugging) over a serial line. Currently only asynchronous-mode SIO drivers are supported.

### 6.5.1 SIO_CHAN and SIO_DRV_FUNCS

Every SIO device is controlled by an **SIO_CHAN** structure. This structure contains a single member, a pointer to an **SIO_DRV_FUNCS** structure. These structures are defined in *installDir*/**vxworks-6.***x*/**target/h/sioLib.h** as:

```
typedef struct sio_chan          /* a serial channel */
    {
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
    } SIO_CHAN;

typedef struct sio_drv_funcs SIO_DRV_FUNCS;

struct sio_drv_funcs             /* driver functions */
    {
    int (*ioctl)
            (
            SIO_CHAN *        pSioChan,
            int               cmd,
            void *            arg
            );

    int (*txStartup)
            (
            SIO_CHAN *        pSioChan
            );
```

```
int (*callbackInstall)
        (
        SIO_CHAN *        pSioChan,
        int               callbackType,
        STATUS            (*callback)(),
        void *            callbackArg
        );

int (*pollInput)
        (
        SIO_CHAN *        pSioChan,
        char *            inChar
        );

int (*pollOutput)
        (
        SIO_CHAN *        pSioChan,
        char outChar
        );
};
```

The members of the **SIO_DRV_FUNCS** structure function as follows:

**ioctl( )**

Points to the standard I/O control interface routine for the driver. This routine provides the primary control interface for any driver. To access the I/O control services for a standard SIO device, use the following symbolic constants:

**SIO_BAUD_SET**, **SIO_BAUD_GET**

Sets and retrieves the port baud rate.

**SIO_HW_OPTS_SET**, **SIO_HW_OPTS_GET**

Sets and retrieves the port hardware options. The available options are: **CLOCAL**, **HUPCL**, **CREAD**, **CSIZE**, **PARENB**, and **PARODD**.

For more information on these options, see *installDir*/**vxworks-6.***x*/**target/h/sioLibCommon.h**.

**SIO_MODE_SET**, **SIO_MODE_GET**, **SIO_AVAIL_MODES_GET**

Sets and retrieves the port mode to switch between polled mode and interrupt driven mode, and find which modes are available. Polled mode is specified as **SIO_MODE_POLL** and interrupt driven mode is specified with **SIO_MODE_INT**. When **SIO_AVAIL_MODES_GET** is used, the values of **SIO_MODE_POLL** and **SIO_MODE_INT** are logically or-d together as follows:

```
*(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
```

**SIO_OPEN**
Sets modem control lines (**RTS** and **DTR**) to **TRUE** if not already set, and initializes the device for user operation. Only valid if **SIO_HUP** is supported.

**SIO_HUP**
Resets **RTS** and **DTR** signals.

Other **ioctl( )** commands can be supported as well. For a more complete list of **ioctl( )** commands that can be supported by serial drivers (such as keyboard modes and keyboard LED states), see *installDir*/**vxworks-6.***x*/**target/h/sioLibCommon.h**.

**txStartup( )**
Provides a pointer to the routine that the system calls when new data is available for transmission. Typically, this routine is called only from the **ttyDrv.o** module. This module provides a level of functionality that allows a raw serial channel to behave with line control and canonical character processing.

**callbackInstall( )**
Provides the driver with pointers to callback routines that the driver can call asynchronously to handle character puts and gets. The driver is responsible for saving the callback routines and arguments that it receives from the **callbackInstall( )** routine. The available callbacks are **SIO_CALLBACK_GET_TX_CHAR** and **SIO_CALLBACK_PUT_RCV_CHAR**.

- Define **SIO_CALLBACK_GET_TX_CHAR** to point to a routine that fetches a new character for output. The driver calls this callback routine with the supplied argument and an additional argument that is the address to receive the new output character (if any). The called routine returns **OK** to indicate that a character was delivered, or **ERROR** to indicate that no more characters are available.
- Define **SIO_CALLBACK_PUT_RCV_CHAR** to point to a routine the driver can use to pass characters to the system. For each incoming character, the callback routine is called with the supplied argument, and the new character as a second argument. Drivers normally do not care about the return value from this call. In most cases, there is nothing that a driver can do but drop a character if the I/O system is not able to receive it.

**pollInput( )** and **pollOutput( )**
Provide an interface to polled mode operations of the driver. These routines are not called unless the device has already been placed into polled mode by an **SIO_MODE_SET** operation.

See *installDir*/**vxworks-6.***x*/**target/src/drv/sio/templateSio.c** for more information on the internal workings of a typical SIO device driver.

### 6.5.2 Polled Mode, WDB, and Kernel Initialization

When WDB is used over a serial channel, it puts the SIO driver into polled mode. This mode disables interrupts and performs I/O operations. Eventually, WDB returns the driver to normal interrupt mode operation.



During BSP development, it is possible to use WDB in polled mode before the kernel is available (see the *VxWorks BSP Developer's Guide: Porting a BSP to Custom Hardware*). In this case, the WDB target agent calls the driver *xxx***ModeSet( )** routine to set the driver into polled mode. Later, the agent puts the driver back into normal interrupt mode. For more information, see the *VxWorks Kernel Programmer's Guide: Kernel*.

Your driver must be able to handle this situation. The WDB agent starts the polled mode session by issuing an **ioctl( )** with the **SIO_MODE_SET** command, which calls the driver *xxx***ModeSet( )** routine. This routine, as well as the polled mode input and output routines, must be able to function without any previous initialization having been performed.

### 6.5.3 Serial Ports, WDB, and Interrupts

SIO driver developers must be aware of two issues related to the use of serial ports for a WDB connection in addition to kernel initialization. These issues are related to interrupts and the order of system initialization.

When using a serial port for a WDB connection, WDB switches the port between polled mode and normal operation, depending on what WDB is doing at any given time. During system mode debugging, which is the only debug mode available during system bringup, WDB puts the serial port into polled mode. But at other times, WDB puts the serial port into normal operation, which usually implies an interrupt-driven mode.

Stray interrupts cause the most serious problem. Connecting an interrupt requires that the system memory pool be available. However, during early parts of system initialization, the system memory pool is not yet available. The driver must wait

until after **usrRoot( )** begins before it can successfully connect an ISR to the device interrupt. The normal calling sequence is:

> **usrRoot( )** => **sysClkConnect( )** => **sysHwInit2( )** => the driver's interrupt initialization routine => **intConnect( )**.

If the driver attempts to connect an ISR before **usrRoot( )** runs, the attempt fails. Any subsequent interrupts are stray interrupts, which cause problems during system initialization.

Another problem is related to the behavior of the actual driver if it attempts to connect interrupts before the system has started. In this case, the SIO driver may not function in interrupt mode thereafter, though it should continue to work in polled mode. As mentioned above, interrupts cannot be connected before the system has started.

One possible workaround for both these problems is to write the SIO driver in such a way that it allows the BSP to signal that interrupts cannot be connected. The generic way to do this is to create a global variable in the driver, indicating whether interrupts can be connected. The value should be initialized to **TRUE**. In the BSP, set the value to **FALSE** early in **sysHwInit( )**, or in the **SYS_HW_INIT_0(I)** macro, if that macro is defined. Then, at the beginning of **sysHwInit2( )**, restore the value to **TRUE**. Using this mechanism, the driver does not need to be modified to run both on a BSP under development and a standard BSP.

# 7 Migrating to VxBus

7.1 Overview 181
7.2 Porting an Existing VxWorks Driver to VxBus 181

## 7.1 Overview

Porting a legacy VxWorks driver to be VxBus compliant involves several simple changes. An overview of the porting process is provided in the steps below. (For more information on VxBus, see *VxWorks Device Driver Developer's Guide, Volume 1: Writing Device Drivers*.)

## 7.2 Porting an Existing VxWorks Driver to VxBus

Porting an existing VxWorks device driver to VxBus generally includes the following steps, briefly mentioned here and discussed in more detail later in the chapter:

1. Verify that the hardware and driver work correctly.
2. Create the VxBus infrastructure required for your driver.

3. Move existing code into the new source file.
4. Remove driver code from the BSP.
5. Add debug code based on conditional compilation.
6. Change the driver initialization over to VxBus.
7. Add the VxBus driver methods required by your driver class.
8. Update names in the source file as necessary.
9. Remove any BSP dependencies.
10. Convert register access in the existing code.
11. Remove all global variables.

### 7.2.1 Verifying Your Hardware and Driver Code

The first step to porting a driver is to ensure that the driver works correctly without VxBus. Starting with a working driver reduces the scope of debugging by limiting errors to the porting process and avoiding problems stemming from the functioning of the original driver.

When you are satisfied that the original driver works correctly, make a backup copy of the driver and your BSP. You can refer back to this copy during the porting process.

### 7.2.2 Creating the VxBus Infrastructure

There are several elements required by every VxBus device driver. Start by adding the empty driver framework that interacts with VxBus. The required parts of this framework include the driver source file itself, one or more optional header files, a CDF file (to allow the driver to be visible to Workbench and the **vxprj** command-line facility), and configuration stub files so that the driver can be included in BSP command-line builds (for more information on these builds, see the *VxWorks Command-Line Tool's User's Guide*).

Once all of the elements of the driver are present in the correct places, configure the BSP for the development effort.

Wind River drivers must be put in the appropriate class-specific directory under *installDir*/**vxworks-6.***x*/**target/src/hwif**. Drivers provided by other vendors must be

put in the driver-specific directory under *installDir*/**vxworks-6.***x*/**target/3rdparty/***vendor*/*driver*.

#### Driver Source File

To create the driver source file, start with a template file or an existing driver from the same driver class. Templates, if available, are kept in the same directory as other drivers of the same class.



#### Driver Header Files (Optional)

Many VxBus device drivers have all source code located in a single source file, with no external header file. However, if your driver includes a number of device-specific macros or other driver-specific information, you can put this information in an optional header file.

When porting an existing driver, you may want to include two header files, one that contains the contents of the existing driver's header file and one that includes the existing driver's BSP-specific stub header file. For example, the pre-VxBus FEI driver used a header file named **fei82557End.h** and a second BSP-specific header file **sysFei82557End.h**. During the VxBus porting process, keep these two files as separate header files. Once the driver nears completion, the files should be consolidated into the existing header file.

#### Driver Component Description File

The component description file (CDF) for your driver allows the driver to be configured and included in a project using standard Wind River tools (Workbench and the **vxprj** command-line utility).

**NOTE:** This section provides an overview of the component description file requirements for adding a driver. For detailed information on CDFs and the component description language, see *VxWorks Kernel Programmer's Guide: Kernel*.

Wind River driver CDF files are located in *installDir*/**vxworks-6.***x*/**target/config/comps/vxWorks** and in the architecture-specific directories under this directory. Third-party driver CDF files are located in *installDir*/**vxworks-6.***x*/**target/3rdparty/***vendor*/*driver*. By convention, driver files use the prefix **40**, for example **40g64120a.cdf**.

In most cases, the CDF file for a driver is simple. You must supply a value for **Component**. Also, the **_INIT_ORDER** value must be set to **hardWareInterFaceBusInit**.

For example:

```
Component   DRV_CLASS_NAME {
    NAME         DriverName
    SYNOPSIS     Description Of Driver
    _CHILDREN    FOLDER_DRIVERS
    REQUIRES     INCLUDE_VXBUS \
                 INCLUDE_PLB_BUS \
                 other requirements
    INIT_RTN     sampleDriverRegister();
    INIT_AFTER   INCLUDE_PLB_BUS
    _INIT_ORDER hardWareInterFaceBusInit
    _CHILDREN    FOLDER_DRIVERS
}
```

Note that by default, the driver is specified as a child of the **FOLDER_DRIVERS** folder. This is done by specifying the **_CHILDREN** option as shown in the example:

```
_CHILDREN    FOLDER_DRIVERS
```

**NOTE:** Be sure to include the leading underscore on the keywords of the CDF file (where shown in the example above). The underscore reverses the meaning. For example, a **_CHILDREN** entry indicates that this component (in this case, your driver) is a child of the specified folder. If the underscore is *not* present, the folder (**FOLDER_DRIVERS**) is configured as a child of your driver, which is not correct.

Many drivers have configuration options. Configuration options that are specified as parameters should be configurable from within Workbench. To do this, provide **Parameter** entries for each parameter and link the parameters to your **Component** with the **CFG_PARAMS** keyword.

For more information on how the driver manages configuration options internally, see *VxWorks Device Driver Developer's Guide, Volume 1*.

### Driver Configuration Stub Files

Configuration stub files provide similar functionality to the CDF file, but are used when building the VxWorks image from the BSP directory using the make command (this is known as the *bspDir*/**config.h** build method).

**NOTE:** In general, you should build your project files using Workbench or the **vxprj** command-line utility. However, the BSP build method described in this section is required in certain development scenarios including early BSP and driver development. For more information on this build method, see the *VxWorks Command-Line Tools User's Guide*.

In most cases, each driver requires two stub files. The stub files are named according to the convention for your driver, with the extensions **.dc** and **.dr**.

The *driverName*.**dc** file usually contains a forward reference to the driver registration routine, and nothing else. Use the Wind River macro **IMPORT** to declare this routine. (Note that all registration routines return a void value.)

The following is a sample driver **.dc** file:

```
IMPORT void sampleDriverRegister(void);
```

The **.dr** file contains a call to the driver registration routine. This call must be surrounded by **#ifdef** and **#endif**. The macro used on the **#ifdef** line must match the component name used in the CDF file (see *Driver Component Description File*, p.183).

The last line must be terminated with a newline (be sure that your editor does not strip it off).

The following is a sample driver **.dr** file:

```
#ifdef DRV_CLASS_NAME
    sampleDriverRegister();
#endif /* DRV_CLASS_NAME */
```

Wind River driver **.dc** and **.dr** files are located in *installDir*/**vxworks-6.***x*/**target/config/comps/src/hwif**. Third-party driver **.dc** and **.dr** files are located in *installDir*/**vxworks-6.***x*/**target/3rdparty/***vendor*/*driver*.

### Modifying the BSP (Optional)

This step is optional because your BSP may already be VxBus-compliant or, depending on the device, there may be no explicit BSP support required (for example, when working with PCI devices).

**NOTE:** Before you start working on your VxBus-enabled driver, you must make sure that your BSP is also VxBus compliant. If your BSP is not enabled for use with VxBus, see the *VxWorks BSP Developer's Guide*.

Depending on the bus type, VxBus may be able to discover your device automatically. For example, when the device is on a PCI bus or variant of PCI, information about the device is available from PCI configuration space. VxBus reads this information and compares it against PCI configuration information provided by a driver for a PCI device. If the information matches, the driver is paired with the device.

However, with the PLB bus type, devices are not discovered automatically. In this case, you must add an entry for your device in the **hcfDeviceList[ ]** array in the BSP **hwconf.c** file.

For easier debugging, configure your BSP so that the show routines are included. Be sure to include the VxBus show routines in addition to the standard show routines. For example, add the following lines in the BSP **config.h** file:

```
#define INCLUDE_SHOW_ROUTINES
#define INCLUDE_VXBUS_SHOW
```

Also include your own driver in **config.h** as follows:

```
#define DRV_CLASS_NAME
```

#### Verifying the infrastructure

Once you have created your driver, compiled it, added it to a library, and configured your BSP, verify that what you have done so far is correct.

To do this, first build the VxWorks image from the BSP directory. Verify that the driver file is included by using the **nm***arch* command and searching for the registration routine.

Next, verify that the CDF file is correct by starting Workbench and configuring the VxWorks image. If everything is correct, your driver should be available in the drivers folder (not greyed out).

Finally, boot the image and run **vxBusShow( )**. Your driver should show up in the list of drivers and the target device should show up in the list of devices.

One common problem—frequently encountered when creating drivers for PLB devices—is that the name of the driver does not match the name you provided in the **hcfDeviceList[ ]** table. When this happens, the output of **vxBusShow( )** displays the entry as an orphan rather than a device. If this happens, you must get the names of the driver and device to match up before proceeding.

VxBus uses the name to match a driver to the hardware. The name is compared using **strcmp( )**. Therefore, the name must be identical (the comparison is case

sensitive). Check that the driver name and the name listed in the **hcfDeviceList[ ]** table in **hwconf.c** are identical and correct as necessary.

The second most common problem at this stage is related to the device's register base address. For PLB devices, the first register base address must be non-null. You can verify this by running **vxBusShow(2)**.

This displays the full set of **pRegBase[ ]** entries for each device (instance and orphan) known by VxBus. If the **pRegBase[0]** entry for your device is zero, correct the problem by supplying the correct base address.



**NOTE:** In some cases, you may not want to supply the register base address in **hwconf.c**. If this is the case for your driver, use a non-null value like **ERROR** or **TRUE** as the register base address value, both of which are non-null. If you choose this option, your driver must not attempt to read or write registers using the VxBus register access mechanism.

Before moving on to the next step, be sure that your device and driver are connected to each other. To do this, lock at the output from **vxBusShow( )**. If the device appears as an orphan, the pairing was not successful.

### 7.2.3 Moving Existing Code into the New Source File

The goal of this phase is to consolidate your existing, working code into the VxBus driver source file.

When the infrastructure for your driver is in place, the next step in porting is to copy the existing driver code into the VxBus driver source file. Note that this includes both the driver proper, and the BSP-specific stub file that you started with.

**NOTE:** Usually, the driver proper and the BSP-specific stub file can go in the same file without trouble. However, you should verify that there are no **LOCAL** routines or **LOCAL** data variables with the same name in the two files. If there are, make whatever modifications are necessary and re-verify the non-VxBus driver.

For this phase, you should modify the CDF file and the **.dc** stub file so that they include the driver source file in the BSP or project compilation. You must do this because many non-VxBus drivers have dependencies on macros that are provided by a BSP file.

### 7.2.4 Removing Driver Code from the BSP

Now, remove all the code relevant to your driver from the BSP. At this point, this code has been copied into the VxBus driver's source file and is no longer required by the BSP.

Once all the driver code is included in the VxBus driver source file and is removed from the BSP, build the BSP with the new driver included. The image should build and boot correctly and the device should work as it did previously. You have now consolidated all of the code to manage the device into a single file. However, you are still using the old driver.

### 7.2.5 Adding Debug Code

After the old driver source code is consolidated into a VxBus driver file, you can add additional debug code. For example, adding debug code is often useful when the driver provides a way to show contents of the driver-specific data area, often referred to as **pDrvCtrl**.

Most drivers benefit by having debug and other diagnostic information available based on a compile-time macro. If the macro is defined, and a flag is set to the desired debug level, debug code is available at runtime.

For example, the following code is a modified version of that done for the **vxbNs16550Sio** driver:

```
#ifdef NS16550_DEBUG_ON
int ns16550vxbDebugLevel = 0;

#ifndef NS16550_DBG_MSG
#define NS16550_DBG_MSG(level,fmt,a,b,c,d,e,f) \
        if ( ns16550vxbDebugLevel >= level )  \
            logMsg(fmt,a,b,c,d,e,f)
#endif /* NS16550_DBG_MSG */

#else /* NS16550_DEBUG_ON */

#define NS16550_DBG_MSG(level,fmt,a,b,c,d,e,f)

#endif /* NS16550_DEBUG_ON */
```

Within the driver, there are many calls to the **NS16550_DBG_MSG( )** macro, such as:

```
NS16550_DBG_MSG(5, "ns16550vxbDevProbe(): INVALID ns16550vxb "
                   "device @ 0x%08x regIndex %d IIR=0x%02x\n",
                   (int)pDev, regBaseIndex, regVal, 4,5,6);
```

This code allows debugging to be disabled entirely by not defining the macro **NS16550_DEBUG_ON** at compile time. In this case, the debug message code—such as the line shown above—is not included in the driver's object module. However, if the macro is defined, the code is included, but not enabled by default. Therefore, to enable the debug messages requires a two-step process. First, compile the driver with **ADDED_CFLAGS=-DNS16550_DEBUG_ON**. Second, after VxWorks has booted, set the **ns16550vxbDebugLevel** variable to a non-zero value to enable all debug messages with a lower debug level value. For example, to enable the debug message shown above, **ns16550vxbDebugLevel** is set to 5 or a greater value.

In addition, it can be helpful to surround diagnostic routines with **#ifdef NS16550_DEBUG_ON** and **#endif /* NS16550_DBG_MSG */**.



**NOTE:** When releasing a driver, much of the debug information used during development continues to be valuable. Therefore, leaving the code in the source file can be beneficial in the future, as long as it can be omitted from the object file. For more information on releasing a driver, see *VxWorks Device Driver Developer's Guide (Vol. 1): Driver Release Procedure*.

The type of debug information that can be added to a driver is discussed in *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

### 7.2.6 Changing Initialization to VxBus

Until this point, the driver is a non-VxBus driver in all important aspects. The first part of the conversion to VxBus is to convert the driver to initialize during VxBus initialization.

These changes are not limited to the driver, but also affect the BSP. Because of the VxBus initialization, the BSP calls to initialize the device are no longer required and should be removed from the BSP.

The driver previously included initialization code that the BSP called directly. The simplest way to convert is to leave the old initialization routine intact, and include a call to it in the **InstInit1( )** or **InstInit2( )** routine referred to by the VxBus registration structure. However, in some cases, moving the code from the old routine into the VxBus initialization routine is cleaner than using a function call.

### 7.2.7 Adding VxBus Driver Methods

Once the VxBus initialization is in place, you can convert the external interface. Usually, this involves finding the VxBus driver methods used by the driver class, searching for routines in the existing driver that provide the required functionality, and creating shim routines that allow the method interface to be used when they are called but resolve to the routines provided by the old driver. Later in the development process, the original code should be copied into what is, at first, a shim layer. When the original code is no longer referenced, delete it. So that the consolidation is not forgotten, make a note in the shim layer that the original code should be consolidated with this layer.

When the functionality used by the required driver methods is available, you can add the methods to the table of methods in your driver and make sure the table is published in the **pMethods** field of **VXB_DEVICE_ID**.

Now test the driver to be sure that it works. After testing the driver by registering with VxBus manually, modify the registration routine so that the driver registers at boot time.

It is not uncommon for device drivers to behave unexpectedly when they are added to the boot process of VxWorks, instead of being started manually. If this occurs, you should inspect the driver's initialization code to make sure that only authorized services are being used at each state of the driver's initialization. For example, **malloc( )** cannot be used until VxBus initialization phase 2, and interrupts cannot be connected until initialization phase 3.

### 7.2.8 Updating Names within the Source File

At this point in the development process, the driver is mostly VxBus compliant, but there are still a few cleanup tasks to complete. The first of these tasks is to update the names of the driver routines. The only required externally-visible symbol is the registration routine. In general, you can change all other routines to **LOCAL**.

Although it is not required, you may wish to change the names of routines and data variables so that they do not clash with the old driver. In certain situations, this step can provide a large advantage. For example, when converting a BSP with a ns16550-compatible console to VxBus, the BSP provides some mechanism to use the console. One conversion strategy is to include both the VxBus **vxbNs16550Sio.c** driver and the BSP code. Then, get a PCI card with an ns16550 port, and set that to the console. Finally, with the PCI card as the console, you can convert the on-board serial devices to use VxBus.

### 7.2.9 Removing BSP Dependencies

Once the source code has been moved into a VxBus driver file, debug information is available, and the driver has an API to be used by the VxBus driver class, it is time to remove BSP dependencies from the driver.

If, as described in *7.2.3 Moving Existing Code into the New Source File*, p.187, you have a modified version of the driver *driverName***.dc** file that causes the source file to be compiled in the context of the BSP, you must change the *driverName***.dc** file so that it does not include the source file in the VxWorks image build.



In order to accomplish this, first compile the driver outside of the BSP, making sure that the driver does not include any BSP header file. By doing this, you can find places in the driver that make use of macros provided by the BSP. These macros need to be resolved by some other method, usually a resource entry or a parameter. When you execute the compile, the BSP-provided macros show up as compile-time warnings of undefined references. Change each of the symbols flagged as undefined references to an entry in the **pDrvCtrl** structure. You also need to fill in the values from a resource or parameter provided by the BSP in **hwconf.c**.

Typically, you should represent the unresolved values as either a resource or as a parameter. Resources are values that are hardware specific. Parameters are values that can be set by the application. You can determine the difference between parameters and resources by testing whether or not the driver continues to run on the same board when the value changes.

If you change the value and the driver continues to function properly, the value is most likely a parameter. If the driver fails to function properly after the change, the value is a resource. You should make this determination for each value. Another test is whether there is a valid default value that works in almost all cases. If so, the value probably represents a parameter.

For example, a prototypical parameter type is the number of transmit buffers in a network interface. A prototypical resource is the frequency of an external timer connected to the device.

Your driver must set each value properly.

For resources, the proper value can be obtained during one of the driver's VxBus initialization routines, which are part of the registration structure. For more information on the registration structure, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*. Make a call to **devResourceGet( )** to obtain the value, and set the global variable according to the value obtained.

For parameters, create a parameter list for the driver, fill in the default values, and put a pointer to the list in the appropriate field of the VxBus registration structure.

Before the first use of this parameter, call either **vxbInstParamByNameGet( )** or **vxbInstParamByIndexGet( )** to retrieve the value.

In addition to fixing undefined macro values, you also need to check external references. Once the file compiles, find undefined symbols using **nm***arch*, review the undefined references, and determine which routines and data are part of the driver. If appropriate, move those routines and variables into the driver and set the value of any data variables using the same methodology described for macro values.

In some cases, it is not appropriate to put certain parts of device management code into the driver. When this happens use one of the following methods:

- When the driver requires certain information that is board-specific, the driver can allow the BSP to provide a routine to fetch that information. The routine is provided to the driver as a resource, of type **HCF_RES_ADDR**. This is treated as a function pointer, and the driver calls that routine to obtain the required information.

  An example of this is when you need to determine the frequency of an external oscillator, and the frequency is not known at compile time. In this case, the BSP must provide a resource—by convention named **clkFreq** and of type **HCF_RES_ADDR**—that is a function pointer that returns the frequency of the external oscillator.
- When the driver requires access to a processor register not available from C, the driver may require that either the architecture code or the BSP provide a routine to access the register, and either call the routine directly or require that a pointer to the routine be provided as an **HCF_RES_ADDR** resource (as described previously).
- When some sections of the driver need to be written in assembly language, the driver may contain inline assembly code, or it may require the BSP provide an **HCF_RES_ADDR** resource (as described previously).

**NOTE:** Due to the complexities of supporting different assembler syntax for different assemblers, and the difficulty of supporting multiple architectures, Wind River does not recommend using inline assembly for general-purpose drivers.

### 7.2.10 Converting Register Access in Existing Code

Many device drivers make direct access to device registers. This can limit the driver to a specific hardware configuration or a specific CPU byte-order.

VxBus provides a register access mechanism that handles byte-order, translation and certain other common register manipulation issues. The routines are described in hardware access section of the *VxWorks Device Driver Developer's Guide (Vol. 3): Device Driver Fundamentals*.

Use of these routines is required in order for your driver to be portable across multiple boards or CPU types.

### 7.2.11 Removing Global Variables



One of the important goals of a generic driver is that it support multiple devices of the same type. Earlier in the development process, you may have chosen to create global variables specific to an instance (that is, a given device and driver paired together). Also, the existing driver you based your development on may have used global variables, perhaps in an array in order to support several devices. These global variables should be removed.

In VxBus, the main identification of a device is the **VXB_DEVICE_ID**. The structure that the **VXB_DEVICE_ID** points to contains a field for **pDrvCtrl**. **pDrvCtrl** is owned by the driver and can be used for any purpose. Most drivers define a structure that contains all instance-specific information.

During initialization, this structure is allocated using **hwMemAlloc( )**, filled in with the data, and a pointer to the structure is saved in the **pDrvCtrl** field. Later, when the driver is called for any reason, the **VXB_DEVICE_ID** is passed as a parameter, from which the driver can extract the **pDrvCtrl** field to get access to the instance-specific data.

In many cases, it is necessary to rewrite the prototype of some routines to pass **pDrvCtrl** or **VXB_DEVICE_ID** as a parameter. This allows each routine within the driver to have access to the information about an instance so that the routines do not need to rely on global variables.

# Index

## Symbols


_CHILDREN 184
_INIT_ORDER 184
_pLinkPoolFuncTbl 36, 37


## A


address resolution, arpresolve( ) 23
APIs, protocol to MUX 17
arpresolve( ) 23
association list 34
ataDrv.c 172
ATAPI drivers 172


## B


BSD 4.3 driver model 9
BSPs
    adding drivers
        required BSP support 4
    config.h 31
    routines
        sysScsiInit( ) 132


## C


CDF 3, 5, 183
    CFG_PARAMS 184
    keywords
        _CHILDREN 184
        _INIT_ORDER 184
        Component 184
        Parameter 184
CDL 5
CFG_PARAMS 184
clkFreq 192
commands
    nmarch 186, 192
commonCmdsTest( ) 138
Component 184
component description file
    *see* CDF
component description language
    *see* CDL
config.h 31
configNet.h 29
configuration stub files 184


## D


DEV_OBJ 61
devResourceGet( ) 191

directCmdsTest( ) 138
directRwTest( ) 138
do_protocol_with_type( ) 16
documentation
about 2

# E

EAGAIN 76
EINVAL 71
EIOCGADDR 72
EIOCGFBUF 72
EIOCGFLAGS 72
EIOCGMIB2 72
EIOCGPOLLCONF 72
EIOCGPOLLSTATS 72
EIOCMULTIADD 72
EIOCMULTIDEL 72
EIOCMULTIGET 72
EIOCPOLLSTART 72
EIOCPOLLSTOP 72
EIOCSADDR 72
EIOCSFLAGS 72
END driver 7
adding a multicast address 76
adding drivers to VxWorks 29
association list 34
backwards compatibility 83
control structure 31
deleting a multicast address 77
driver responsibilities 14
entry points 10
error conditions 84
fair access bounding 40, 43
forming an address for packet transmission 78
getting a data-only mBlk 79
getting the multicast address table 78
handling a polled receive 75
handling a polled send 74
implementing the generic MIB interface 86
interface to VxWorks 24
interrupt handlers 26
interrupt masking 28
interrupt re-enabling 40, 47
launching your driver 24
loading a device 66
mBlk structure 64
memory resources 31
MUX responsibilities 14
network layer to data link layer address resolution 23
performance 84
protocol responsibilities 14
providing a control interface 71
receive and transmit descriptors 32
receive handler interlocking flag 40, 44
receive loop 40, 41
receiver stall handling 40, 46
required entry points 65
required structures 58
returning addressing information 80
sending data out on the device 72
setting up a memory pool 36
starting a loaded driver 73
status dump routines 91
stopping a loaded driver 74
support for scatter-gather 52
transmit descriptor clean routine 54
transmit-packet-complete handler interlocking flag 52
two-tiered polling 40, 47
unloading a device 68
END driver components 12
END_ERR_BLOCK 85
END_ERR_DOWN 21, 85
END_ERR_FLAGS 85
END_ERR_INFO 21, 85
END_ERR_NO_BUF 85
END_ERR_RESET 21, 85
END_ERR_UP 21, 85
END_ERR_WARN 21, 85
END_IFCOUNTERS 90
END_IFDRVCONF 90
END_LOAD_FUNC 30
END_LOAD_STRING 30
END_OBJ 58
END_OBJ_INIT 62
END_RCV_RTN_CALL 43
endAddressForm( ) 66, 78

endDevTbl[ ] 29
endEtherPacketAddrGet( ) 80
endIoctl( ) 65, 71
endLoad( ) 24, 30, 50, 58, 63, 65, 66
endM2Free( ) 88
endM2Init( ) 87
endM2Ioctl( ) 87
endM2Packet( ) 87
endMCastAddrAdd( ) 61, 65, 76
endMCastAddrDel( ) 61, 66, 77
endMCastAddrGet( ) 61, 66, 78
endPacketAddrGet( ) 66
endPacketDataGet( ) 66, 79
endPollReceive( ) 66, 75
endPollSend( ) 66, 74
endReceive( ) 39
endSend( ) 65, 72
endStart( ) 65, 73
endStop( ) 65, 74
endTbl 30
endUnload( ) 65, 68
enhanced network driver
    *see* END driver
ENOSPC 71
ENOTSUP 71
etherInputHook( ) 16
etherMultiLib 76
Ethernet driver 7
    *see also* END driver
etherOutputHook( ) 16

## F

files
    hwconf.c 186
FOLDER_DRIVERS 184

## H

hardware timers, characteristics 144
hardWareInterFaceBusInit 184
HCF_RES_ADDR 192
hwconf.c 186

## I

IFF_ALLMULTI 60
IFF_BROADCAST 59
IFF_DEBUG 59
IFF_LINK0 60
IFF_LINK1 60
IFF_LINK2 60
IFF_LOAN 60
IFF_LOOPBACK 60
IFF_MULTICAST 60
IFF_NOARP 60
IFF_NOTRAILERS 60
IFF_OACTIVE 60
IFF_POINTOPOINT 60
IFF_PROMISC 60
IFF_RUNNING 60
IFF_SCAT 60
IFF_SIMPLEX 60
IFF_UP 59
INCLUDE_END 31
INCLUDE_NET_INIT 29
INCLUDE_NETWORK 29
INCLUDE_TIMESTAMP 147
INCLUDE_USER_TIMESTAMP 147
initialization
    VxBus 189
interrupt handlers 26
interrupt masking 28

## L

legacy driver 1
libraries
    etherMultiLib 76
    netBufLib 25, 35
    SCSI 98
    scsi2Lib 97
    scsiCommonLib 97, 101

scsiCtrlLib 100
scsiDirectLib 97, 100
scsiMgrLib 98
scsiSeqLib 98, 101
LL_HDR_INFO 63

# M

macros
END_OBJ_INIT 62
END_RCV_RTN_CALL 43
INCLUDE_NET_INIT 29
INCLUDE_NETWORK 29
mBlk 64
memory drivers 174
migrating
adding debug code 188
adding VxBus driver methods 190
CDF 183
converting register access 192
creating VxBus infrastructure 182
header files 183
LOCAL routines and data variables 187
modifying the BSP 185
moving existing code into a new source file 187
removing
BSP dependencies 191
driver code from the BSP 188
global variables 193
to VxBus 181
updating names in the source file 190
verifying
driver code 182
VxBus infrastructure 186
VxBus initialization 189
MULTI_TABLE 78
multi-mode serial drivers 176
*see also* SIO drivers
multiplexer
*see* MUX
MUX
defined 8
entry points 10
MUX API, interactions with 17
MUX_PROTO_OUTPUT 15
MUX_PROTO_PROMISC 15, 16
MUX_PROTO_SNARF 15
muxAddressForm( ) 16, 18
muxAddrResFuncAdd( ) 19
muxAddrResFuncDel( ) 19
muxAddrResFuncGet( ) 19
muxBind( ) 10, 14, 17
muxDataPacketGet( ) 18
muxDevLoad( ) 10, 17, 24, 30, 39
muxDevStart( ) 17, 24
muxDevStop( ) 18
muxDevUnload( ) 18
muxError( ) 84
muxIoctl( ) 18
muxMCastAddrAdd( ) 18
muxMCastAddrDel( ) 18
muxMCastAddrGet( ) 18
muxPacketAddrGet( ) 18
muxPacketDataGet( ) 16, 18
muxReceive( ) 16, 18, 43
muxSend( ) 16, 18
muxShutdown( ) 18
muxTxRestart( ) 18
muxUnbind( ) 10, 17, 18

# N

NET_FUNCS 63, 65
NET_PROTOCOL 17, 19
netBufLib 25, 35
netJobAdd( ) 26, 28, 44
netJobRing 27, 44, 52
netMblkToBufCopy( ) 75
netPoolCreate( ) 25, 35, 37
netPoolInit( ) 35
netPoolRelease( ) 70
netTupleGet( ) 42
nmarch 186, 192

# O

one-shot timer 143

# P

packets, handling
    reception 39
    transmission 52
Parameter 184
pDrvCtrl 188
periodic interrupt timer 143
pMethods 190
porting
    a legacy driver to the VxBus model 181
    an END driver from another OS 83
project facility
    CDF entries 5
    CDL 5
protocol data structure 19

# Q

quiescent state 4

# R

receive handler interlocking flag 40, 44
return value
    EINVAL 71
    ENOSPC 71
    ENOTSUP 71
RFC 1213 86
RFC2233 86
routines
    arpresolve( ) 23
    devResourceGet( ) 191
    endM2Free( ) 88
    endM2Init( ) 87
    endM2Ioctl( ) 87
    endM2Packet( ) 87
    etherInputHook( ) 16
    etherOutputHook( ) 16
    muxAddressForm( ) 16, 18
    muxAddrResFuncAdd( ) 19
    muxAddrResFuncDel( ) 19
    muxAddrResFuncGet( ) 19
    muxBind( ) 10, 14, 17
    muxDataPacketGet( ) 18
    muxDevLoad( ) 10, 17, 24, 30, 39
    muxDevStart( ) 17, 24
    muxDevStop( ) 18
    muxDevUnload( ) 18
    muxError( ) 84
    muxIoctl( ) 18
    muxMCastAddrAdd( ) 18
    muxMCastAddrDel( ) 18
    muxMCastAddrGet( ) 18
    muxPacketAddrGet( ) 18
    muxPacketDataGet( ) 16, 18
    muxReceive( ) 16, 18, 43
    muxSend( ) 16, 18
    muxShutdown( ) 18
    muxTxRestart( ) 18
    muxUnbind( ) 10, 17, 18
    netJobAdd( ) 26, 28, 44
    netPoolCreate( ) 25, 35, 37
    netPoolInit( ) 35
    netPoolRelease( ) 70
    netTupleGet( ) 42
    scsiCacheSnoopDisable( ) 113
    scsiCacheSnoopEnable( ) 113
    scsiCacheSynchronize( ) 113
    scsiCtrlInit( ) 102, 112
    scsiDiskTest( ) 137
    scsiDiskThruputTest( ) 137
    scsiIdentMsgBuild( ) 113
    scsiIdentMsgParse( ) 113
    scsiMgrBusReset( ) 112
    scsiMgrCtrlEvent( ) 112
    scsiMgrEventNotify( ) 102, 112
    scsiMgrThreadEvent( ) 112
    scsiMsgInComplete( ) 112
    scsiMsgOutComplete( ) 112
    scsiMsgOutReject( ) 113

scsiSpeedTest( ) 139
scsiSyncXferNegotiate( ) 112
scsiThreadInit( ) 113
scsiTransact( ) 97
scsiWideXferNegotiate( ) 112
stackError( ) 20
stackRcvRtn( ) 10, 19
stackShutdownRtn( ) 10, 21, 22
stackTxRestartRtn( ) 10, 22
sysHwInit( ) 4
sysInByte( ) 82
sysInLong( ) 82
sysInWord( ) 82
sysOutByte( ) 82
sysOutLong( ) 82
sysOutWord( ) 82
sysScsiInit( ) 132
sysTimestamp( ) 165, 166
sysTimestampConnect( ) 163
sysTimestampDisable( ) 164
sysTimestampEnable( ) 164
sysTimestampFreq( ) 165
sysTimestampLock( ) 166, 167
sysTimestampPeriod( ) 164
sysTimestampRoutine( ) 167
tapeFsTest( ) 139
usrNetInit( ) 29
vxbInstParamByIndexGet( ) 192
vxbInstParamByNameGet( ) 192
vxBusShow( ) 186
wdDelete( ) 70
wvTmrRegister( ) 147

# S

scatter-gather 52
SCSI
commands 97
common access library 101
controller libraries 100
direct access library 100
module layout 95
objects and data structures 95
sequential access library 101
SCSI drivers 93
advanced controller driver example 113
advanced I/O processor example 127
basic controller example 103
BSP interface 132
data coherency problems 136
development 135
programming interface 101
sysScsi.c template 133
template 103
test suites 136
VxWorks interface 98
SCSI manager 98
SCSI_PHYS_DEV 97
SCSI_TRANSACTION 97
scsi2Lib 97
scsi2Lib.h 112
scsiCacheSnoopDisable( ) 113
scsiCacheSnoopEnable( ) 113
scsiCacheSynchronize( ) 113
scsiCommonLib 97, 101
scsiCtrlInit( ) 102, 112
scsiCtrlLib 100
scsiDirectLib 97, 100
scsiDiskTest( ) 137
scsiDiskThruputTest( ) 137
scsiIdentMsgBuild( ) 113
scsiIdentMsgParse( ) 113
scsiMgrBusReset( ) 112
scsiMgrCtrlEvent( ) 112
scsiMgrEventNotify( ) 102, 112
scsiMgrLib 98
scsiMgrThreadEvent( ) 112
scsiMsgInComplete( ) 112
scsiMsgOutComplete( ) 112
scsiMsgOutReject( ) 113
scsiSeqLib 98, 101
scsiSpeedTest( ) 139
scsiSyncXferNegotiate( ) 112
scsiThreadInit( ) 113
scsiTransact( ) 97
scsiWideXferNegotiate( ) 112
show routines
VxBus 186

SIO drivers 176
*see also* multi-mode serial drivers
polled mode 179
SIO_CHAN 176
SIO_DRV_FUNCS 176
sioLib.h 176
stackError( ) 10, 20
stackRcvRtn( ) 10, 19
stackShutdownRtn( ) 10, 21, 22
stackTxRestartRtn( ) 10, 22
sysDev.c 4
sysHwInit( ) 4
sysInByte( ) 82
sysInLong( ) 82
sysInWord( ) 82
sysLib.c 4
sysOutByte( ) 82
sysOutLong( ) 82
sysOutWord( ) 82
sysScsi.c
sysScsiInit( ) 132
template 133
sysScsiInit( ) 132
sysSerial.c 4
System Viewer 166
sysTimestamp( ) 165, 166
sysTimestampConnect( ) 163
sysTimestampDisable( ) 164
sysTimestampEnable( ) 164
sysTimestampFreq( ) 165
sysTimestampLock( ) 166, 167
sysTimestampPeriod( ) 164
sysTimestampRoutine( ) 167

## T

tapeFsTest( ) 139
tasks
tNet0 24
tNetTask 24, 26, 28
tUsrRoot 24
templateSio.c 179
timers, hardware, characteristics of 144
timestamp drivers 141
BSP interface 163
components 148
configuration 163
sample drivers 148
VxWorks interface 146
VxWorks requirements 145
working with the System Viewer 147
timestamp timer 143
tNet0 24
tNetTask 24, 26, 28
*see also* tNet0
transmit-packet-complete handler interlocking flag 52
TS_SKEW 169
tuple defined 25
tuple, memory pool 34
tUsrRoot 24

## U

unloading
an END driver 69
usrNetInit( ) 29

## V

VXB_DEVICE_ID 190
vxbInstParamByIndexGet( ) 192
vxbInstParamByNameGet( ) 192
VxBus 2
configuration stub files 184
creating VxBus infrastructure 182
driver source file 183
header files 183
porting a legacy driver to 181
vxBusShow( ) 186
VxWorks
components 183

# W

WDB agent 15
WDB_COMM_END 15
wdDelete( ) 70
Wind River System Viewer 141, 166
writing
    a new END driver 81
wvTmrRegister( ) 147