VxWorks

Guide to Writing and Porting Device Drivers

Order Number: AA–QLZKA–TE

This manual provides information about writing and porting VxWorks device drivers.

Revision/Update Information: This is a new manual.
Operating System and Version: DEC OSF/1, Version 2.0 or higher
Software Version: VxWorks for Alpha, Version 3.1A

Digital Equipment Corporation
Maynard, Massachusetts
Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.


All Rights Reserved.

The following are trademarks of Digital Equipment Corporation: DEC, DECchip, DEC OSF/1, DECwindows, RT-11, ThinWire, and the Digital logo.

The following are third-party trademarks:

Intel is a trademark of Intel Corporation.
Motorola and 68000 are registered trademarks of Motorola, Inc.
MS-DOS is a registered trademark of Microsoft Corporation.
NCR is a registered trademark of NCR Corporation.
NFS and Sun are registered trademarks of Sun Microsystems, Inc.
OSF/1 is a registered trademark of the Open Software Foundation.
POSIX is a registered trademark of IEEE.
UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Ltd.
VxWorks is a registered trademark of Wind River Systems, Inc.

All other trademarks and registered trademarks are the property of their respective holders.
Contents

Preface .............................................................................. xi

Part I Introduction to VxWorks Device Drivers

1 Device Driver Overview
   1.1 Device Driver Environment ........................................ 1–1
   1.1.1 The Hardware ................................................... 1–1
   1.1.1.1 Buses ...................................................... 1–2
   1.1.1.2 Device Controllers ........................................ 1–3
   1.1.1.3 Peripheral Devices ......................................... 1–3
   1.1.1.4 Bus Adapters ............................................... 1–3
   1.1.2 The Software .................................................. 1–4
   1.1.2.1 VxWorks Kernel ........................................ 1–4
   1.1.2.2 VxWorks I/O System ................................. 1–4
   1.1.2.3 Local File Systems ...................................... 1–5
   1.2 Device Driver Structure ......................................... 1–6
   1.2.1 Nonblock Device Drivers ................................. 1–6
   1.2.2 Block Device Drivers ...................................... 1–7
   1.3 Porting an Existing VxWorks Device Driver .............. 1–7

2 VxWorks for Alpha Target Platforms
   2.1 Supported Microprocessors and Single-Board Computers ... 2–2
   2.2 Supported Buses .................................................. 2–3
3 VxWorks for Alpha I/O Support

3.1 I/O Software Architecture ........................................ 3–1
3.2 Issues Affecting I/O Programming ............................... 3–2
  3.2.1 Data Granularity .............................................. 3–3
  3.2.2 Scatter/Gather Mapping ..................................... 3–3
  3.2.3 Read/Write Ordering ......................................... 3–4
  3.2.4 Write Buffer ................................................... 3–4

Part II Writing VxWorks Device Drivers

4 Designing the Device Driver

  4.1 Identifying Driver Requirements ................................ 4–1
  4.2 Making Design Decisions ........................................ 4–2
    4.2.1 Block versus Nonblock Devices ............................ 4–2
    4.2.2 I/O Modes .................................................. 4–3
    4.2.3 Timeouts and Multiple Accesses ........................... 4–4
  4.3 Nonstandard Devices ............................................ 4–4

5 Programming the Device Driver

  5.1 Driver Support Libraries ....................................... 5–2
  5.2 Data Structures .................................................. 5–3
  5.3 Required I/O Functions ......................................... 5–4
    5.3.1 Nonblock Device Functions ................................. 5–4
      5.3.1.1 Creating a File ....................................... 5–6
      5.3.1.2 Removing a File ....................................... 5–7
      5.3.1.3 Opening a File ........................................ 5–7
      5.3.1.4 Closing a File ......................................... 5–8
      5.3.1.5 Reading from a File ................................... 5–9
      5.3.1.6 Writing to a File ...................................... 5–9
      5.3.1.7 Performing Control Functions ......................... 5–10
    5.3.2 Block Device Functions .................................... 5–10
      5.3.2.1 Reading Blocks from the Device ..................... 5–11
      5.3.2.2 Writing Blocks to the Device ......................... 5–13
      5.3.2.3 Performing Control Functions ....................... 5–15
      5.3.2.4 Resetting the Device .................................. 5–18
      5.3.2.5 Checking Device Status ................................ 5–19
  5.4 Implementing select() ............................................ 5–20
  5.5 Mapping the Bus .................................................. 5–23
    5.5.1 Mapping Base, Granularity, and Offset .................. 5–24
    5.5.2 Pointer I/O ................................................ 5–24
6 Writing a Serial Device Driver

6.1 Driver Overview ......................................... 6–2
6.2 Driver Data Structures .................................. 6–4
6.3 Driver Operations ...................................... 6–6
tyCoDevCreate() ........................................... 6–7
tyCoDrv() .................................................. 6–9
tyCoInt() .................................................. 6–9
tyColct() ............................................... 6–10
tyCoOpen() ........................................... 6–11
tyCoStartup() ......................................... 6–12
6.4 Sample Serial Device Driver .......................... 6–13
## 7 Writing a Network Interface Driver

### 7.1 Driver Overview ........................................... 7–4
#### 7.1.1 Attaching the Driver to the System .................... 7–5
#### 7.1.2 Interrupt Handling ................................ 7–5
#### 7.1.3 Support Routines ................................ 7–6
#### 7.1.4 Ethernet Hook Routines ............................. 7–6
#### 7.1.5 System-Dependent Network Support Routines ........... 7–7
### 7.2 Driver Data Structures ................................ 7–7
### 7.3 Driver Operations ..................................... 7–10

- `attach( )` ........................................ 7–12
- `Init( )` ............................................. 7–14
- `Int( )` ............................................. 7–15
- `Ioctl( )` ............................................ 7–16
- `Output( )` ........................................ 7–17
- `Reset( )` ........................................... 7–18
- `Watchdog( )` ...................................... 7–18

### 7.4 Network Support Routines ............................... 7–20

- `bcopy_from_mbufs( ), copy_from_mbufs( )` ................... 7–21
- `bcopy_to_mbufs( ), copy_to_mbufs( )` ....................... 7–22
- `build_cluster()` .................................... 7–23
- `check_trailer()` ..................................... 7–25
- `do_protocol(), do_protocol_with_type()` .................... 7–26
- `ether_attach()` ..................................... 7–28
- `ether_output()` ..................................... 7–29
- `IF_DEQUEUE()` ..................................... 7–30
- `IF_ENQUEUE()` ..................................... 7–31
- `netJobAdd()` ...................................... 7–32
- `set_if_addr()` ..................................... 7–33
- `splnet(), splimp()` ................................... 7–34
- `splx()` ............................................ 7–35

### 7.5 Sample Network Interface Driver ........................ 7–36
8 Writing a SCSI Device Driver

8.1 Driver Overview .............................................. 8–3
8.2 Driver Data Structures ........................................ 8–7
8.3 Driver Operations ............................................. 8–14
  xxxBusPhaseGet() ............................................. 8–17
  xxxBytesIn() .................................................. 8–18
  xxxBytesOut() .................................................. 8–19
  xxxCtrlCreate() ............................................... 8–20
  xxxCtrlInit() ................................................... 8–22
  xxxDevSelect() .................................................. 8–23
  xxxDmaBytesIn() ............................................... 8–24
  xxxDmaBytesOut() ............................................. 8–25
  xxxIntr() ....................................................... 8–26
  xxxMsgInAck() .................................................. 8–27
  xxxScsiBusReset() ............................................... 8–28
  xxxScsiTransact() ............................................. 8–28
  xxxSelTimeOutCvt() ............................................ 8–30
  xxxShow() ....................................................... 8–31
8.4 Driver Support Macros ........................................ 8–32
  SCSI_DEBUG_MSG(), SCSI_INT_DEBUG_MSG() .................. 8–32
8.5 Sample SCSI Device Drivers .................................. 8–34
  8.5.1 NCR 53C90 ASC SCSI Driver Listing .................. 8–34
  8.5.2 NCR 53C810 PSIOP SCSI Driver Listing ............... 8–62

Part III Porting Existing VxWorks Device Drivers to the Alpha Platform

9 Porting Considerations

9.1 Data Representation ............................................ 9–2
9.2 Data Access ................................................... 9–2
9.3 Data Alignment ................................................. 9–3
9.4 Constants ....................................................... 9–3
  9.4.1 Assignment and Argument Passing ....................... 9–4
9.4.2 Shift Operations ............................................ 9–5
9.5 Variables ....................................................... 9–5
  9.5.1 Declarations ............................................... 9–5
  9.5.2 Assignments and Function Arguments ................... 9–5
  9.5.3 The sizeof Operator ....................................... 9–7
9.5.4 Pointer Subtraction .................................. 9–7
9.6 Structures .................................................. 9–7
9.6.1 Size ...................................................... 9–7
9.6.2 Alignment ............................................. 9–8
9.6.2.1 Member Alignment .................................. 9–8
9.6.2.2 Alignment Guidelines .............................. 9–8
9.6.2.3 Alignment Examples ............................... 9–9
9.6.3 Bit Fields .............................................. 9–9
9.7 Pointers ................................................... 9–10
9.8 File Systems ............................................... 9–11
9.9 Unavailable Instructions ................................. 9–11
9.10 Using the lint Command ................................. 9–11

10 Using the 68K VMEbus Library Interface

10.1 systLib Routines .......................................... 10–1
10.2 New I/O Routines ......................................... 10–2
10.3 Probing the Bus .......................................... 10–3
10.4 Connecting to Bus Interrupts ............................ 10–4
10.5 Mapping the Bus .......................................... 10–4
10.5.1 Outbound Mapping .................................... 10–4
10.5.2 Inbound Mapping ...................................... 10–5
10.6 68K VMEbus Interface Reference ........................ 10–5
     sysBusIntGen() ........................................... 10–6
     sysBusTas() .............................................. 10–6
     sysBusToLocalAdrs() .................................... 10–7
     sysIntDisable() ......................................... 10–9
     sysIntEnable() .......................................... 10–9
     sysLocalToBusAdrs() .................................... 10–10
     sysVmeMemProbe() ..................................... 10–11
     sysVmeReadByte() ...................................... 10–12
     sysVmeReadInt() ........................................ 10–13
     sysVmeReadQuad() ...................................... 10–14
     sysVmeReadWord() ...................................... 10–15
     sysVmeWriteByte() ..................................... 10–16
     sysVmeWriteInt() ....................................... 10–17
     sysVmeWriteQuad() ..................................... 10–18
     sysVmeWriteWord() ..................................... 10–19
## 11 Driver Porting Example

11.1 Porting Example Using the \texttt{adpLib} Library .................... 11–2
11.2 Porting Example Using the 68K VMEbus Interface ............. 11–9

### Index

#### Examples

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5–1</td>
<td>Abbreviated Nonblock Driver Code</td>
</tr>
<tr>
<td>5–2</td>
<td>Block Device Read Routine</td>
</tr>
<tr>
<td>5–3</td>
<td>Block Device Write Routine</td>
</tr>
<tr>
<td>5–4</td>
<td>Block Device I/O Control Routine</td>
</tr>
<tr>
<td>5–5</td>
<td>Block Device Reset Routine</td>
</tr>
<tr>
<td>5–6</td>
<td>Block Device Status Check Routine</td>
</tr>
<tr>
<td>5–7</td>
<td>Driver Support for \texttt{select()}</td>
</tr>
<tr>
<td>5–8</td>
<td>Device Driver Interrupt Service Routine</td>
</tr>
<tr>
<td>5–9</td>
<td>Connecting to an Interrupt</td>
</tr>
<tr>
<td>5–10</td>
<td>DMA Processing</td>
</tr>
<tr>
<td>5–11</td>
<td>Nonblock Device Driver Initialization Routine</td>
</tr>
<tr>
<td>5–12</td>
<td>Nonblock Device Initialization Routine</td>
</tr>
<tr>
<td>5–13</td>
<td>Block Device Driver Initialization Routine</td>
</tr>
<tr>
<td>5–14</td>
<td>Block Device Initialization Routine</td>
</tr>
<tr>
<td>6–1</td>
<td>Z8C530 Serial Driver Data Structure \texttt{TY_CO_DEV}</td>
</tr>
<tr>
<td>6–2</td>
<td>Serial Driver Data Structure \texttt{TY_DEV}</td>
</tr>
<tr>
<td>6–3</td>
<td>Z8C530 Serial Driver Array Structure \texttt{tyCoDv}</td>
</tr>
<tr>
<td>6–4</td>
<td>PC87312 Serial Driver Array Structure \texttt{tyCoDv}</td>
</tr>
<tr>
<td>6–5</td>
<td>Z8C530 Serial Driver Listing</td>
</tr>
<tr>
<td>7–1</td>
<td>DECchip 21040 Ethernet Driver Data Structure \texttt{drv_ctrl}</td>
</tr>
<tr>
<td>7–2</td>
<td>Network Driver Data Structure \texttt{arpcom}</td>
</tr>
<tr>
<td>7–3</td>
<td>Network Driver Data Structure \texttt{ifnet}</td>
</tr>
<tr>
<td>7–4</td>
<td>DECchip 21040 Ethernet Driver Listing</td>
</tr>
<tr>
<td>8–1</td>
<td>\texttt{ncr5390Lib} SCSI Driver Data Structure \texttt{NCR_5390_SCSI_CTRL}</td>
</tr>
<tr>
<td>8–2</td>
<td>\texttt{ncr810Lib} SCSI Driver Data Structure \texttt{NCR810_SCSI_CTRL}</td>
</tr>
<tr>
<td>8–3</td>
<td>SCSI Driver Data Structure \texttt{SCSI_CTRL}</td>
</tr>
<tr>
<td>8–4</td>
<td>SCSI Driver Data Structure \texttt{SCSI_PHYS_DEV}</td>
</tr>
</tbody>
</table>
VxWorks for Alpha is a Digital UNIX software product for developing dedicated realtime applications that run on a supported set of Alpha and Motorola 68K processor platforms.

Manual Objectives

This manual describes how to write a VxWorks device driver for an Alpha target platform. It also describes how to port an existing VxWorks device driver to an Alpha system from a system based on the Motorola 68000 family.

This manual is intended to supplement the VxWorks Programmer's Guide.

To use this manual, you must have purchased and installed VxWorks Realtime Tools and Board Support Package (BSP) software. If you have purchased a VxWorks Board Porting Kit or Source Kit, additional network and SCSI driver templates are available to you.

Intended Audience

This manual is for the programmer or system engineer who is already familiar with VxWorks concepts, as presented in the VxWorks Programmer's Guide, and with the C programming language. The audience is assumed to have experience with UNIX operating systems and program development tools.

Document Structure

This manual is organized in three parts, as follows:

- Part I, Introduction to VxWorks Device Drivers, provides an overview of VxWorks device drivers. This part contains the following chapters:
  - Chapter 1, Device Driver Overview
  - Chapter 2, VxWorks for Alpha Target Platforms
  - Chapter 3, VxWorks for Alpha I/O Support
• Part II, Writing VxWorks Device Drivers, describes how to write a VxWorks device driver. This part contains the following chapters:
  • Chapter 4, Designing the Device Driver
  • Chapter 5, Programming the Device Driver
  • Chapter 6, Writing a Serial Device Driver
  • Chapter 7, Writing a Network Interface Driver
  • Chapter 8, Writing a SCSI Device Driver

• Part III, Porting Existing VxWorks Device Drivers to the Alpha Platform, describes how to port an existing VxWorks device driver to an Alpha platform. This part contains the following chapters:
  • Chapter 9, Porting Considerations
  • Chapter 10, Using the 68K VMEbus Library Interface
  • Chapter 11, Driver Porting Example

Conventions

The following conventions are used in this manual:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>The default user prompt is the user’s system name followed by a right angle bracket. In this manual, a percent sign (%) represents this prompt.</td>
</tr>
<tr>
<td>#</td>
<td>A number sign is the default superuser prompt.</td>
</tr>
<tr>
<td>-&gt;</td>
<td>This is the VxWorks shell prompt for interactive input.</td>
</tr>
<tr>
<td><strong>bold type</strong></td>
<td>Bold type is used to indicate new terms defined in the text. Bold type is also used for the case-sensitive name of a library, driver, command, or command option, when it appears in text or in syntax displays. For example:</td>
</tr>
<tr>
<td></td>
<td>The adpLib library includes the adpRead() routine.</td>
</tr>
<tr>
<td><strong>keyword</strong></td>
<td>In text, this typeface indicates the case-sensitive name of a routine, function, pathname, directory, or file. For example:</td>
</tr>
<tr>
<td></td>
<td>The adpRead() routine is provided by the adpLib library.</td>
</tr>
<tr>
<td>Convention</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>italic type</td>
<td>Italic typeface indicates variable information, such as user-supplied information in commands, syntax, or example text. For example:</td>
</tr>
<tr>
<td></td>
<td>The year, month, and day arguments specify the current date.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Square brackets, in a command syntax, enclose optional items. For example:</td>
</tr>
<tr>
<td></td>
<td>% shadow [-x directory] old-path [new-path]</td>
</tr>
<tr>
<td>. . .</td>
<td>Horizontal ellipsis points indicate that the preceding item can be repeated one or more times. It is used in syntax descriptions and function definitions.</td>
</tr>
<tr>
<td>. . .</td>
<td>Vertical ellipsis points indicate that a portion of an example or figure is omitted.</td>
</tr>
<tr>
<td>UPPERCASE lowercase</td>
<td>The system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.</td>
</tr>
<tr>
<td>cat(1)</td>
<td>Cross-references to Digital UNIX online reference pages include the appropriate section number in parentheses. For example, a reference to cat(1) indicates that you can find the material on the Digital UNIX command cat in Section 1 of the Digital UNIX online reference pages.</td>
</tr>
<tr>
<td>Key</td>
<td>In tables, key names are designated by a box. For example:</td>
</tr>
<tr>
<td></td>
<td>Return.</td>
</tr>
<tr>
<td>Ctrl/x</td>
<td>This construct indicates a control key sequence. Press the key labeled Ctrl while you simultaneously press another key. For example: Ctrl/C.</td>
</tr>
<tr>
<td>yourCloneTree</td>
<td>This is shorthand for the location to which you copied or cloned the installed directory structure. For example, if you cloned the VxWorks directory tree to /usr/users/fred/work, a documentation reference to yourCloneTree/config/mv167/system.st refers to /usr/users/fred/work/config/mv167/system.st.</td>
</tr>
</tbody>
</table>
Associated Documents

For information about installing VxWorks for Alpha, see the VxWorks Installation Guide.

The following documents provide platform-independent coverage of VxWorks concepts:

• VxWorks Release Notes
• Introduction to VxWorks
• VxWorks Installation Guide
• VxWorks Programmer’s Guide
• VxWorks Reference Manual
• VxWorks POSIX Programming Guide
• VxWorks POSIX Reference Manual
• VxWorks POSIX Information Concerning P1003.4
• VxWorks POSIX Information Concerning Draft 4 of P1003.4a
• VxWorks Guide to VxGDB
• VxWorks Guide to DECwindows
Part I

Introduction to VxWorks Device Drivers

Part I presents an overview of VxWorks for Alpha device drivers. It includes a description of the VxWorks for Alpha environment and the general structure of a device driver. It also includes a brief description of the supported Alpha platforms and buses, as well as a description of the Alpha architecture issues that affect how you program a device driver.

This part consists of the following chapters:

• Chapter 1, Device Driver Overview
• Chapter 2, VxWorks for Alpha Target Platforms
• Chapter 3, VxWorks for Alpha I/O Support
Device Driver Overview

VxWorks for Alpha applications, and realtime applications in general, are typically time-critical and must respond quickly and predictably to events generated by external devices. A realtime system is normally dedicated to one application, with all hardware and software system resources focused on that application.

One integral part of a realtime system is the set of device drivers used to interface with peripheral devices.

This chapter describes the following topics:

• The VxWorks for Alpha device driver environment, Section 1.1
• The structure of a VxWorks for Alpha device driver, Section 1.2
• The programming considerations involved in porting an existing VxWorks device driver to VxWorks for Alpha, Section 1.3

1.1 Device Driver Environment

The VxWorks for Alpha device driver environment consists of specific hardware and software components. Sections 1.1.1 and 1.1.2 describe the hardware and software components, respectively.

1.1.1 The Hardware

Figure 1–1 shows the principal hardware components in a typical realtime environment. The components include:

• Buses, Section 1.1.1.1
• Device controllers, Section 1.1.1.2
• Peripheral devices, Section 1.1.1.3
• Bus adapters, Section 1.1.1.4

1.1.1 Buses

A **bus** is the physical transmission medium that connects computer system components — the CPU, memory, and peripheral devices — and provides a communications path for addresses, data, and control information. Each bus is an implementation of a bus architecture that specifies the physical characteristics of the medium — including logic signals, timings, and connectors — and defines the communications protocol for sending and receiving data.

The I/O configuration in Figure 1-1 includes two buses: an internal **communications bus** and an external **I/O bus**. The communications bus is the physical transmission medium for communications between the processor and memory. The I/O bus is an external transmission medium that devices share and use to communicate with the processor.

The Alpha architecture allows expansion of I/O devices by permitting cascaded multiple buses through tightly coupled interconnects. For systems with more than one I/O bus, the bus closest to the CPU is known as the **primary** (or **local**) bus; all other buses are known as **secondary** buses.
Chapter 2 presents an overview of microprocessor platforms and bus architectures supported by VxWorks for Alpha.

1.1.1.2 Device Controllers

A **device controller** provides the hardware interface between a bus and an external device. Sometimes a controller handles several devices. In other cases, a controller is built into the device.

1.1.1.3 Peripheral Devices

A **peripheral device** is a hardware component that connects to a device controller and receives, transmits, or stores data.

Each peripheral device has one or more **control/status registers (CSRs)** that the driver uses to communicate with the device. A driver might read and write a device's CSRs to issue commands or check the device's status and use other registers to exchange data.

Each device CSR is assigned a physical address. Although CSRs reside in the external devices, they are treated the same as locations in physical memory containing integer data. A device driver issues commands to a peripheral device by writing to the device's registers as if they were physical memory locations. Similarly, the driver obtains status information about the device by reading its CSRs. That portion of a system's physical address space allocated to device registers is called **I/O space**.

There are two fundamental types of devices that can be accessed through the VxWorks I/O system: **block** and **nonblock**. Block devices are random-access devices where data is transferred in blocks. The I/O system considers all other devices as nonblock (or character) devices. Examples of block devices include hard disks and diskettes. Examples of nonblock devices include serial devices, such as terminals and line printers, and data acquisition devices, such as digitizers and scanners.

1.1.1.4 Bus Adapters

A **bus adapter** is a communications device that provides the physical and logical connection between two separate buses. The adapter can transmit and receive data according to the protocol of either bus, allowing devices on an I/O bus to communicate with a driver program running on a system processor.

In addition to providing a physical connection and protocol conversion, the adapter must support the types of I/O operations that its associated buses support.
Some devices perform **direct memory access (DMA)** I/O transfers, accessing memory directly without generating an interrupt for each atomic transfer. To support DMA transfers, a bus adapter must contain **mapping registers** that associate the physical addresses of a transfer buffer in memory with physical bus addresses used by the device. (A map that associates contiguous bus addresses with noncontiguous areas of memory is called a **scatter/gather map**.)

Scatter/gather mapping is also implemented on some buses for programmed I/O (PIO). In these circumstances, the bus adapter must contain another set of registers to map I/O memory space to addresses on the bus.

### 1.1.2 The Software

The VxWorks for Alpha software environment in which a device driver operates includes the following components:

- The VxWorks kernel, Section 1.1.2.1
- The VxWorks I/O system, Section 1.1.2.2
- The local file system, Section 1.1.2.3

#### 1.1.2.1 VxWorks Kernel

VxWorks for Alpha is a multitasking environment that allows realtime applications to be constructed as sets of independent tasks, each with its own thread of execution and set of system resources. The VxWorks **kernel** is that portion of the system software that schedules task execution and coordinates the use of system resources.

The kernel uses interrupt-driven, priority-based task scheduling, featuring fast context switches and low interrupt latency. Any C subroutine can be spawned as a separate task, with its own context and stack. Task control allows tasks to be suspended, resumed, deleted, delayed, and moved in priority.

The VxWorks kernel provides several types of traditional semaphores for intertask synchronization. It also provides a message queue facility, intertask pipes, sockets and signals for fast and flexible intertask communication.

#### 1.1.2.2 VxWorks I/O System

The VxWorks I/O system offers uniform device-independent access to many kinds of devices.

As in the Digital UNIX operating system environment, VxWorks applications access devices through the I/O system by opening named files. A file can be one of two things:

- An unstructured device, such as a serial communications channel
A logical file on a structured, random-access device

I/O requests can be made at two different levels — basic and buffered. The basic VxWorks I/O interface is source compatible with the I/O primitives in the standard C library: `creat()`, `remove()`, `open()`, `close()`, `read()`, `write()`, and `ioctl()`.

The VxWorks buffered I/O package is compatible with the `stdio` package in Digital UNIX.

When responding to a user I/O request, the VxWorks I/O system does minimal processing before control is given to the device driver. Instead, the I/O system simply acts as a switch to route user requests to the proper driver. The driver has complete control over how the requests are serviced. If a device implements standard protocols, VxWorks provides several high-level subroutine libraries the driver can use to process a request. However, if throughput is critical or if the device does not fit a standard model, the driver is free to implement different protocols, device-specific functions, and even different file systems without interference from the I/O system itself.

For more information about the structure and use of the VxWorks I/O system, refer to the VxWorks Programmer’s Guide.

1.1.2.3 Local File Systems

VxWorks for Alpha includes support for several different local file systems for use with block devices. Block devices use a standard interface so that file systems can be freely mixed with device drivers.

The VxWorks I/O architecture makes it possible to use two or more different local file systems concurrently by assigning them to different partitions of the same physical disk device.

VxWorks for Alpha offers three local file systems:

- `dosFs` — compatible with MS-DOS for personal computers
- `rt11Fs` — compatible with the file system of the RT-11 operating system
- `rawFs` — a simple, raw disk file system that treats an entire disk partition as a single large file

In VxWorks for Alpha, the file system is not tied to a device or its driver. A device can be associated with any file system. Alternate, user-supplied file systems can be written to use existing drivers by following the standard interface between the file system, the driver, and the VxWorks I/O system.
1.2 Device Driver Structure

Under VxWorks for Alpha, a device driver consists of one or more program threads that process user requests for I/O from a device. There is generally one device driver for each type of device in the system configuration.

1.2.1 Nonblock Device Drivers

A driver for a nonblock device implements seven basic I/O functions for the device:

- `creat()`
- `remove()`
- `open()`
- `close()`
- `read()`
- `write()`
- `ioctl()`

In general, a nonblock device driver has routines that implement each of these functions, although some routines may be omitted if the function is nonoperative with the device. Typically, the `creat()` and `remove()` functions are nonoperative for nonblock devices.

When a user makes an I/O request, the VxWorks I/O system routes the request to the appropriate routine of the associated driver. The driver routine runs in the context of the calling task, just as if it had been called directly from the user program. Thus, the driver is free to use any facilities normally available to tasks, including I/O to other devices. (This means most drivers must use some mechanism to provide mutual exclusion to critical regions of code.)

In addition to the routines that implement the seven basic I/O functions, nonblock device drivers must also supply three other routines:

- An initialization routine that installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization.
- A routine that adds the devices to be serviced by the driver to the I/O system.
- Interrupt-level routines to process the interrupts generated by the devices.
1.2.2 Block Device Drivers

Drivers for block devices interface with a file system, rather than directly with the I/O system. The file system, in turn, implements most I/O functions. The driver need only supply routines to read and write blocks, reset the device, perform I/O control, and check device status.

(Block device drivers also have a number of other special requirements, described in detail in Part II.)

1.3 Porting an Existing VxWorks Device Driver

VxWorks for Alpha is written for target platforms based on Digital’s 64-bit Alpha architecture. Alpha single-board computers (SBCs) can provide a substantial increase in performance over other SBCs, such as the Motorola 68K family. However, to take advantage of this performance increase, you must be aware of certain differences between Alpha SBCs and other platforms that affect how you program VxWorks for Alpha device drivers.

Programming differences arise because of the following features of the Alpha architecture and the implementation of Alpha SBCs:

• The 64-bit Alpha architecture forces changes in the way data is represented and aligned.

• The Alpha architecture supports a minimum data granularity and alignment of 32 bits.

• As a true RISC architecture, the Alpha architecture does not provide certain processor instructions typically found under a CISC architecture.

• In addition to a fixed, direct mapping of physical address space to I/O address space, Alpha SBCs also permit a flexible, software-controlled mapping.

• On Alpha CPUs, the order of read and write operations can be different from the order specified in the program source code.

• Alpha CPUs have a write buffer.

These differences affect general programming, as well as programming device drivers.

Chapter 3 describes the I/O programming issues that arise due to the Alpha architecture. Chapter 9 describes general issues of programming in the Alpha 64-bit environment.
VxWorks for Alpha is for target platforms based on Digital’s 64-bit Alpha architecture. This chapter briefly describes the single-board computers (SBCs) and buses supported by the VxWorks software.

Table 2-1 summarizes the Alpha target platforms supported by VxWorks for Alpha.

<table>
<thead>
<tr>
<th>Board/CPU Chip</th>
<th>Primary Bus</th>
<th>Secondary Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPvme</td>
<td>PCI (integrated)</td>
<td>VMEbus</td>
</tr>
<tr>
<td>DECchip 21066</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECchip 21068</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AXPpci</td>
<td>PCI (integrated)</td>
<td>ISA</td>
</tr>
<tr>
<td>DECchip 21066</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECchip 21068</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB64</td>
<td>ISA (pin bus interface)</td>
<td>—</td>
</tr>
<tr>
<td>DECchip 21064</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB64+</td>
<td>PCI (host bridge)</td>
<td>ISA</td>
</tr>
<tr>
<td>DECchip 21064</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECchip 21064A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EB66</td>
<td>PCI (integrated)</td>
<td>ISA</td>
</tr>
<tr>
<td>DECchip 21066</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECchip 21068</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.1 Supported Microprocessors and Single-Board Computers

VxWorks for Alpha supports the following Alpha microprocessors:

- DECchip 21064
- DECchip 21064A
- DECchip 21066
- DECchip 21068

Board support packages (BSPs) are available for the following production single-board computers:

- AXPvme family
  
  The AXPvme family consists of the following SBCs:
  
  - AXPvme 160, based on the DECchip 21066 microprocessor
  - AXPvme 64, based on the DECchip 21068 microprocessor

  These SBCs use the Peripheral Component Interconnect (PCI) bus as their primary, on-board bus, and include an 8-bit SCSI-2 controller, a DMA Ethernet controller, and a 32-bit VMEbus master/slave interface with VME64 and DMA capability. They also support an IEEE P1386.1 PCI mezzanine card (PMC) with one option slot.

  A low-cost variant of the AXPvme 64 (AXPvme 64LC) is also supported without the SCSI-2 controller.

- AXPpci 33
  
  The AXPpci 33 is driven by either the DECchip 21066 or the DECchip 21068 microprocessor. The CPU chip includes an integrated controller to the primary, on-board PCI bus. The PCI bus provides one 8-bit SCSI-2 controller and two 32-bit PCI expansion slots. A PCI-to-ISA bridge provides an interconnect to an Industry Standard Architecture (ISA) secondary bus with four 16-bit ISA expansion slots. An additional, dual usage (PCI or ISA) slot is also provided.

In addition to these production SBCs, VxWorks for Alpha supports the following evaluation boards for use as hardware and software development platforms:

- EB64
  
  The EB64 is based on the DECchip 21064 microprocessor. It includes a pin bus interface to an Industry Standard Architecture (ISA) bus with two serial ports, a floppy disk controller, an on-board Ethernet controller, and two ISA expansion slots.
• **EB64+**

The EB64+ is based on either the DECchip 21064 or the DECchip 21064A microprocessor. It uses the PCI bus as its primary, on-board bus, and includes an 8-bit SCSI-2 controller, a DMA Ethernet controller, and one dedicated PCI expansion slot. A PCI-to-ISA bridge provides an interconnect to an ISA secondary bus with two dedicated ISA expansion slots. An additional, dual usage (PCI or ISA) slot is also provided.

• **EB66**

The EB66 is driven by either the DECchip 21066 or the DECchip 21068 microprocessor. The CPU chip includes an integrated controller to the primary, on-board PCI bus. The PCI bus provides one 8-bit SCSI-2 controller, a DMA Ethernet controller, and one dedicated PCI expansion slot. A PCI-to-ISA bridge provides an interconnect to an ISA secondary bus with two dedicated ISA expansion slots. An additional, dual usage (PCI or ISA) slot is also provided.

### 2.2 Supported Buses

**VxWorks for Alpha** supports the following bus architectures:

- **PCI Bus**

  The peripheral component interconnect (PCI) bus is a high-performance 32-bit bus with multiplexed address and data lines. It is used as the primary bus for all supported VxWorks for Alpha target platforms except the EB64 evaluation board. The PCI bus provides an interconnect to high-performance Ethernet and SCSI-2 controllers (for those SBCs that support them), along with one or more expansion slots for additional I/O devices.

  The EB64+ evaluation board connects to the PCI bus through a DECchip 21071-DA PCI host bridge. On the AXPvme boards, the AXPpci 33 and the EB66, the PCI interface is integrated into the CPU chip.

- **ISA Bus**

  The Industry Standard Architecture (ISA) bus used on Alpha targets platforms conforms to the industry-standard IEEE-PI996 16-bit bus architecture. It is used as the primary bus on the EB64 evaluation board, connecting through a pin bus interface. On the AXPpci 33 SBC and the EB66 and EB64+ evaluation boards, the ISA bus is used as a secondary bus, connecting through an Intel SIO PCI-to-ISA bridge.
As a secondary bus, the ISA bus supports the following:

- A combination controller providing a floppy disk controller and two universal asynchronous receiver/transmitters (UARTs)
- Time of year (TOY) clock
- 8K bytes of nonvolatile ram (NVRAM)
- One or more dedicated ISA slots for additional I/O devices

**VMEbus**

The VMEbus is a standard backplane interface conforming to the open VME (Versa Module Eurocard) bus architecture defined by the IEEE Standard 1014. Used as a secondary bus on the AXPvme family of SBCs, the VMEbus connects to the PCI primary bus through a VIC64 PCI-to-VMEbus bridge providing a full 32-bit VMEbus master/slave interface with VME64 and DMA capabilities.
This chapter presents an overview of how the VxWorks software implements I/O support for Alpha target platforms. It describes the following topics:

- The internal architecture of the I/O support software, Section 3.1
- I/O programming issues arising from the Alpha architecture, Section 3.2

3.1 I/O Software Architecture

The VxWorks for Alpha software manages I/O buses in a way that is transparent to application software. At system startup, an adapter data structure is established for each I/O bus in the system. For systems with more than one I/O bus, initialization code creates a hierarchy of adapter structures reflecting parent/child relationships between the existing buses.

As shown in Figure 3–1, the bus architecture within the VxWorks for Alpha software consists of two layers:

- High-level hardware-independent routines called by user code
- Low-level bus-specific interface routines that reside in the board support package (BSP) for the target platform

The high-level layer of this software architecture consists of a set of adapter interface routines that operate independently from platform type. These routines (found in the adpLib library) transparently handle the complexities of such things as multilevel interrupt dispatching, as well as mapping of device interface registers for DMA and programmed I/O transfers.
3.2 Issues Affecting I/O Programming

Architectural features of Alpha target platforms affect the way device drivers must manage I/O buses and devices. In particular, the following features impact I/O programming:

- The Alpha architecture permits a minimum data granularity and alignment of 32 bits.
- In addition to fixed, direct mapping of physical address space to I/O space, Alpha CPUs also permit a flexible, software-controlled mapping.
- On Alpha CPUs, the order of read/write operations can be different from the order specified in the program source code.
• Alpha CPUs have a write buffer.

The following sections describe the implications of each of these features.

Note that the adpLib routines are designed to handle all the programming issues described in this section.

3.2.1 Data Granularity

The Alpha architecture supports a minimum data granularity and alignment of 32 bits. This feature was designed to enhance performance, but obviously has a significant impact on 8-bit or 16-bit I/O. To accommodate this type of I/O, Alpha processors implement two types of I/O addressing.

When the I/O bus can be accessed with a 32-bit or 64-bit data granularity, you can use the traditional one-to-one mapping of I/O space — each physical processor address maps to an I/O address on the bus. This is known as dense-space addressing.

However, when accessing the bus requires an 8-bit or 16-bit data granularity, you must use a form of I/O addressing known as sparse-space addressing. Sparse-space addressing works by encoding the data type into the address. The processor performs each bus read or write as a 32-bit access, but maps the desired data to the appropriate address range within the 32-bit value.

Since accessing each address on the bus requires a 32-bit access, regardless of the actual data size, each 64-bit processor address contains one or more invalid byte addresses that do not map to any bus location. Hence the name sparse-space addressing.

To use sparse-space addressing, your program must calculate the sparse-space processor address that maps to the required bus address. Then depending on the data type, the data must be shifted to or from the appropriate byte within the 32-bit value. The adpLib routines adpRead() and adpWrite() perform these manipulations automatically.

3.2.2 Scatter/Gather Mapping

32-bit VxWorks targets, such as the Motorola 68K family of SBCs, provide a direct mapping of processor addresses to bus addresses for DMA and PIO data transfers. This mapping is configured in the hardware and is transparent to the user.

In contrast, the address mapping provided by Alpha targets is dynamically configurable by the software and consists of many separate mappings of smaller blocks. This type of mapping is known as scatter/gather mapping. Some Alpha targets also have directly mapped buses.
You can use \texttt{adpLib} library routines to handle both types of mapping. Use \texttt{adpCreateMapPath()} and \texttt{adpFreeMapPath()} to create and free a dynamic mapping. For directly mapped buses, \texttt{adpCreateMapPath()} returns the map base and size.

\subsection*{3.2.3 Read/Write Ordering}
Under the Alpha architecture, read and write accesses are not necessarily executed in the sequence specified in the program source code. If two memory or hardware accesses are “unordered” (as determined by a predefined set of rules), the CPU is permitted to rearrange those accesses for higher performance.

This reordering can have undesirable side effects when writing device CSRs. For example, if your driver must load the device data register before setting the “go” bit in the device control register, reordering those two writes will obviously produce incorrect results.

To ensure that two read/write operations occur in the order in which they are specified in your program, you must insert a memory barrier instruction (MB) between the two references. The \texttt{adpWrite()} routine does this by following the write operation with a call to the \texttt{mb()} function which invokes the MB instruction.

\subsection*{3.2.4 Write Buffer}
One design feature of the Alpha architecture is the use of a write buffer for write operations to both memory and to the I/O bus. The write buffer does the following:

- Merges write operations of consecutive 32-bit values into a single write of a 64-bit value.
- Collapses successive write operations to the same address into a single write operation, using data from the last write.
- Holds write operations for some specific amount of time, waiting to see if subsequent write operations can be merged or collapsed. (This is often referred to as write posting.)

While data is in the write buffer, the processor continues as if the write operation has already occurred.

Using a write buffer can buy obvious performance gains when writing to memory. However, undesirable side effects can occur when writing to an I/O register if the write is to be followed by a read of the same register.
To circumvent write buffer activity, the write buffer can be flushed after a write operation. The `adpWrite()` routine does this by including a call to the `mb()` function which invokes the memory barrier (MB) instruction.
Part II describes how to write a VxWorks device driver. In addition to addressing specific programming issues, this part includes examples of drivers for different types of devices.

This part consists of the following chapters:

- Chapter 4, Designing the Device Driver
- Chapter 5, Programming the Device Driver
- Chapter 6, Writing a Serial Device Driver
- Chapter 7, Writing a Network Interface Driver
- Chapter 8, Writing a SCSI Device Driver
This chapter describes how to design your device driver and contains the following topics:

- Section 4.1, Identifying Driver Requirements
- Section 4.2, Making Design Decisions
- Section 4.3, Nonstandard Devices

### 4.1 Identifying Driver Requirements

The first step in designing your device driver is to identify the driver requirements. In particular, you need to identify general and specific requirements concerning driver structure, I/O modes, and driver communication and synchronization.

The following list of questions will help you identify design issues that apply to your driver:

- Which device features need to be implemented?
- Does the device impose any timing constraints?
- Is the device a block device that supports random accessing of data?
- Can the device perform DMA transfers?
- Does the application need to change device parameters dynamically at runtime?
- Does the driver need to handle I/O for multiple device units?
- Does the application need to be able to specify a timeout for I/O from the device?
- Does the application need to be able to wait for I/O from more than one device at a time?
- What device registers does the driver need to access?
• Does the device need to access system memory?
• Does the driver need to communicate or synchronize with another driver?

### 4.2 Making Design Decisions

Once you identify your driver requirements, you can use the information to make decisions about the design of your driver.

#### 4.2.1 Block versus Nonblock Devices

The VxWorks I/O system permits access to two fundamental types of devices: block and nonblock. A block device is a random-access device organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk.

From the point of view of the VxWorks I/O system, all other devices are nonblock devices.

As shown in Figure 4–1, block devices have a slightly different interface to the I/O system than nonblock devices. Rather than interacting directly with the I/O system, block device drivers interact with a file system. The file system, in turn, interacts with the I/O system.

Block and nonblock device drivers differ in the following ways:

- A driver for a nonblock device implements seven basic I/O functions: `creat()`, `remove()`, `open()`, `close()`, `read()`, `write()`, and `ioctl()`. In general, a nonblock device driver has a separate routine for each of these functions, although some of the routines may be omitted if the function is nonoperative for the device. Typically, the `creat()` and `remove()` functions are omitted for nonblock devices.

- A driver for a block device supplies routines to read and write blocks of data, perform I/O control, and check the device status. In addition, block device drivers have a number of other special requirements.

These differences are described in detail throughout Chapter 5.
4.2.2 I/O Modes

VxWorks supports two modes of I/O: programmed I/O (PIO) and direct memory access (DMA).

PIO transfers are single access. That is, one piece of data is read or written during each operation. Program execution is typically suspended until the transfer is complete. Programmed I/O is also known as **outbound** I/O because during the transfer, the device driver is the bus master.

DMA, on the other hand, permits the device to transfer large amounts of data directly to or from system memory without suspending program execution. Many devices that support DMA transfers use a set of scatter/gather registers in the bus adapter. These registers map system memory addresses to the bus addresses used by the device. DMA transfers are known as **inbound** I/O because during the transfer, the device is the bus master.
If your device supports DMA, you should consider the transfer rate of the device and the size of the data packets to be accessed before you decide to implement this I/O mode. DMA is more appropriate for larger transfers.

Section 5.8 describes how to program your driver to execute DMA transfers.

### 4.2.3 Timeouts and Multiple Accesses

To allow calling tasks to specify a timeout for the device or to wait for input from multiple devices, you may want your driver to support the `select()` function.

The `select()` function allows calling tasks to do the following:

- Specify a timeout to wait for I/O from the device (for example, time out on a UDP socket if the packet never arrives)
- Simultaneously wait for I/O from multiple devices of the same type (for example, multiple pipes used for different data priorities)
- Simultaneously wait for I/O from multiple devices of different types (for example, a server task using both pipes and sockets)

Section 5.4 describes how to implement the `select()` function.

### 4.3 Nonstandard Devices

Not all devices fit neatly into the VxWorks I/O system architecture. Some real-time devices, in particular, are not easily classified as either block or nonblock devices. A real-time clock, for example, is certainly not a block device. But to try to force it into a nonblock device mold would result in virtually all its functionality being handled by the `xxIoctl()` routine — at a loss of efficiency in both programming and performance.

Other devices are associated with well-defined, special-purpose protocols, such as TCP/IP or SCSI, which require coding of driver routines not defined in the basic VxWorks I/O system model. Examples are network interface devices (Chapter 7) and SCSI devices (Chapter 8).

If your device does not fit the standard VxWorks I/O system model, or if your driver needs more direct access due to timing considerations, you are free to write your driver as a set of utility routines that are invoked directly by a calling task.

However, try as much as possible to use the VxWorks for Alpha adapter interface provided by `adpLib`. Doing so will maintain the portability of your driver code across Alpha platforms. Even though your driver will not be registered with the I/O system, you should consider using `adpLib` routines for functions such as the following:
• Connecting to interrupts (adpIntConnect())
• Enabling interrupts (adpSetBusIntEnable())
• Mapping for outbound I/O (adpCreateMapPath())
• Reading or writing bus addresses (adpRead() and adpWrite())
• Obtaining adapter information (adpLookup())
• Obtaining device-specific information (adpFindDevice())

An example of a nonstandard, yet adpLib-compliant, device driver can be found in yourCloneTree/src/demo/XYCOM/xycom_driver0.c. This driver is for the XYCOM 203 realtime clock and is a good example of a driver that performs low-level I/O while maintaining compatibility with the adpLib standard for Alpha bus access.
This chapter describes the basic steps for programming your VxWorks device driver. This information is intended mainly as a summary and companion guide to the following manuals:

- The VxWorks Programmer’s Guide describes the components of a VxWorks for Alpha system and gives basic guidelines for their use.
- The VxWorks Reference Manual describes the system libraries and routines provided with the VxWorks software, including details of their calling syntax and return values.
- The hardware supplement for your target system.
- The technical specification of your device.

When you program your driver, you may need to do the following:

1. Set up data structures, Section 5.2
2. Program the required I/O functions, Section 5.3
3. Implement the `select()` function, Section 5.4
4. Map device addresses on the bus, Section 5.5
5. Handle interrupts, Section 5.6
6. Communicate or synchronize with applications or other devices, Section 5.7
7. Set up the driver for DMA transfers, Section 5.8
8. Program routines to install your driver and device in the I/O system, Section 5.9
9. Handle bus-specific programming issues, Section 5.10
10. Build the driver and load it into the system image, Section 5.11
Many of the programming examples throughout this chapter are taken from the driver for the National Semiconductor Corporation PC87311/PC87312 floppy disk controller that runs on the Industry Standard Architecture (ISA) bus. For the complete source listing of this driver, see the yourCloneTree/src/demo/fdc directory.

5.1 Driver Support Libraries

The adpLib I/O library provides routines that support handling of I/O through adapter data structures. An adapter data structure is established for each I/O bus on your target system and is used to manage the bus in a way that is transparent to application-level software. Routines available in the adpLib library permit device drivers to perform the following operations for a device on a supported bus; not all operations apply to all buses:

- Reset the bus
- Configure a node on the bus
- Set and clear bus interrupt enables
- Post, check, and clear interrupts
- Perform a read-modify-write operation on the bus
- Probe the bus for a specified read or write address
- Read or write to a bus address
- Create and release hardware data paths that map the address space between two buses
- Load scatter/gather map registers for inbound or outbound I/O
- Connect an interrupt service routine written in C to an interrupt vector
- Access node information structures
- Obtain a pointer to an adapter data structure
- Update or obtain interrupt vector information

VxWorks for Alpha provides several other subroutine libraries that may help you write your device driver. See the VxWorks Reference Manual and the corresponding library manpage entries for more detail.
5.2 Data Structures

Most data structures used by the VxWorks I/O system are created automatically when drivers are installed and devices are added. However, your driver must supply a structure known as a **device descriptor** for each device it supports. A device descriptor consists of a required header, followed by optional device-specific information as required by the driver. This information can include items such as the device address, buffers, semaphores, and so on.

The format of a device descriptor differs for block and nonblock devices.

- A device descriptor for a nonblock device must begin with a structure called a device header (DEV_HDR). This structure contains the device name string and the number of the driver that will service the device. Note, however, that the driver does not need to fill in the device header, only any device-specific information that may follow it.

  The device header is filled in when the device is added to the I/O system through a call to routine `iosDevAdd()` from within the device creation routine. (See Section 5.9.)

- A device descriptor for a block device must begin with a logical block device structure known as a BLK_DEV. This structure describes the device in a generic fashion, specifying only those common characteristics that must be known to a file system being used with the device. Fields within the BLK_DEV specify various physical configuration variables for the device, as well as the address of the driver routines used to manipulate the device. Table 5–1 shows the fields of the BLK_DEV structure.

  The device descriptor is generally allocated and initialized within the device creation routine. Unlike nonblock device drivers, however, the device creation routine for a block device does not call `iosDevAdd()`. Instead, it returns the address of the device descriptor. This address is then passed to the device initialization routine of the appropriate file system. (See Section 5.9.)

Your device driver may need to allocate other data structures, as required by the system library routines it uses.
Table 5–1  Fields in the BLK_DEV Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bd_blkRd</td>
<td>Address of driver routine that reads blocks from the device.</td>
</tr>
<tr>
<td>bd_blkWrt</td>
<td>Address of driver routine that writes blocks to the device.</td>
</tr>
<tr>
<td>bd_ioctl</td>
<td>Address of driver routine that performs device I/O control.</td>
</tr>
<tr>
<td>bd_reset</td>
<td>Address of driver routine that resets the device (NULL if none).</td>
</tr>
<tr>
<td>bd_statusChk</td>
<td>Address of driver routine that checks disk status (NULL if none).</td>
</tr>
<tr>
<td>bd_removable</td>
<td>TRUE if device is removable; otherwise FALSE.</td>
</tr>
<tr>
<td>bd_nBlocks</td>
<td>Total number of blocks on the device.</td>
</tr>
<tr>
<td>bd_bytesPerBlk</td>
<td>Number of bytes per block.</td>
</tr>
<tr>
<td>bd_blksPerTrack</td>
<td>Number of blocks per track.</td>
</tr>
<tr>
<td>bd_nHeads</td>
<td>Number of heads (surfaces).</td>
</tr>
<tr>
<td>bd_retry</td>
<td>Number of times to retry failed reads or writes.</td>
</tr>
<tr>
<td>bd_mode</td>
<td>Device mode (write protect status). Generally set to UPDATE.</td>
</tr>
<tr>
<td>bd_readyChanged</td>
<td>TRUE if device ready status has changed. Should be</td>
</tr>
<tr>
<td></td>
<td>initialized to TRUE to cause the disk to be mounted.</td>
</tr>
</tbody>
</table>

5.3 Required I/O Functions

All VxWorks for Alpha device drivers are expected to implement a basic set of I/O functions. A major difference between block and nonblock device drivers is the set of functions required.

5.3.1 Nonblock Device Functions

In general, a driver for a nonblock device implements seven basic I/O routines for the device. These routines correspond to the seven basic I/O primitives available in the standard C library, stdio. A routine may be omitted if the corresponding function is nonoperative for the device.

- xxCreat() — create a file
- xxRemove() or xxDelete() — delete a file
- xxOpen() — open a file (optionally, create a file)
- xxClose() — close a file
- xxRead() — read from a previously created or opened file
• **xxWrite()** — write to a previously created or opened file
• **xxIoctl()** — perform a control function on the file or device

---

**Note**

Remember that under VxWorks for Alpha, applications access I/O devices by opening named files. A file can refer to either an actual file on a structured random-access device or raw data on an unstructured device. Since nonblock devices, by definition, do not contain file structures, the **xxCreat()** and **xxRemove()** functions are typically omitted for nonblock device drivers.

Example 5–1 shows the abbreviated code for the basic I/O functions of a hypothetical driver for a serial device. Note that within the VxWorks for Alpha I/O system, each driver has a short, unique name (such as “net” or “tyCo”) that is prefixed to each of its routine names — the xx in the list above. The name of the example driver in Example 5–1 is “nblk.”

**Example 5–1 Abbreviated Nonblock Driver Code**

```c
/**********************************************************************
* The following routines implement the basic I/O functions. Note
* that the return value type is driver dependent.
*/

DEVICE *nblkOpen (nblkDev, remainder, mode)

DE\[1\]CE *nblkDev; /* address of device descriptor block */
char *remainder; /* file name */
int mode; /* open mode */

/* serial devices should have no file name part */
if (remainder[0] != 0)
    return (ERROR);
else
    return (nblkDev);

int nblkRead (nblkDev, buffer, nBytes)

DEVICE *nblkDev; /* device descriptor block */
char *buffer; /* address of data buffer */
int nBytes; /* number of bytes to read */

...

int nblkWrite (nblkDev, buffer, nBytes)

... (continued on next page)
```
Example 5–1 (Cont.) Abbreviated Nonblock Driver Code

int nblkIoctl (nblkDev, requestCode, arg)

Your driver routines must conform to the calling syntax of each I/O function, as shown in the following subsections. Beyond that, you are free to implement your driver in any way appropriate. Since these routines will run in the calling task's context, your driver is free to use any facilities normally available to tasks, including I/O to other devices.

5.3.1.1 Creating a File

The calling syntax for the xxCreat() routine is as follows:

DEVICE *xxCreat
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *fname, /* file name */
    int mode /* open mode */
}

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver's device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type DEV_HDR. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>fname</td>
<td>Specifies a pointer to the file name portion of the file descriptor as supplied by the calling task.</td>
</tr>
<tr>
<td>mode</td>
<td>Specifies the mode in which the created file should be opened.</td>
</tr>
</tbody>
</table>

The routine should return any value that identifies the file — typically a pointer to the device descriptor. If the operation failed, the routine should return a NULL pointer.

Note
The creat() function is typically omitted for nonblock devices.
5.3.1.2 Removing a File

The calling syntax for the `xxRemove()` (or `xxDelete()`) routine is as follows:

```c
STATUS xxRemove
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *fname /* file name */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver's device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type DEV_HDR. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>fname</td>
<td>Specifies a pointer to the file name portion of the file descriptor as supplied by the calling task.</td>
</tr>
</tbody>
</table>

The routine should return OK if the operation was successful; otherwise, it should return ERROR.

--- Note ---

The `remove()` function is typically omitted for nonblock devices.

5.3.1.3 Opening a File

The calling syntax for the `xxOpen()` routine is as follows:

```c
DEVICE *xxOpen
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *fname, /* file name */
    int mode /* open mode */
}
```

where the routine parameters have the following meanings:
**Parameter** | **Meaning**
--- | ---
\texttt{xxDEV} | Specifies a pointer to the driver's device descriptor structure. Note that the typedef \texttt{DEVICE} specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type \texttt{DEV_HDR}. (See Section 5.2 for a description of the device descriptor.)
\texttt{fname} | Specifies a pointer to the file name portion of the file descriptor as supplied by the calling task.
\texttt{mode} | Specifies the mode in which the file should be opened.

The routine should return any value that identifies the file — typically a pointer to the device descriptor. If the operation failed, the routine should return a NULL pointer.

### 5.3.1.4 Closing a File

The calling syntax for the \texttt{xxClose()} routine is as follows:

```c
STATUS xxClose
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *fname /* file name */
}
```

where the routine parameters have the following meanings:

**Parameter** | **Meaning**
--- | ---
\texttt{xxDEV} | Specifies a pointer to the driver's device descriptor structure. Note that the typedef \texttt{DEVICE} specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type \texttt{DEV_HDR}. (See Section 5.2 for a description of the device descriptor.)
\texttt{fname} | Specifies a pointer to the file name portion of the file descriptor as supplied by the calling task.

The routine should return \texttt{OK} if the operation was successful; otherwise, it should return \texttt{ERROR}.
5.3.1.5 Reading from a File

The calling syntax for the `xxRead()` routine is as follows:

```c
int xxRead
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *buf,    /* address of data buffer */
    int  nBytes  /* number of bytes to read */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver’s device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type DEV_HDR. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>buf</td>
<td>Specifies a pointer to the buffer to receive the data read.</td>
</tr>
<tr>
<td>nBytes</td>
<td>Specifies the number of bytes to read.</td>
</tr>
</tbody>
</table>

The routine should return the number of bytes actually read.

5.3.1.6 Writing to a File

The calling syntax for the `xxWrite()` routine is as follows:

```c
int xxWrite
{
    DEVICE *xxDEV, /* address of device descriptor */
    char *buf,    /* address of data buffer */
    int   len     /* length of data buffer */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver’s device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type DEV_HDR. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>buf</td>
<td>Specifies a pointer to the buffer containing the data to be written.</td>
</tr>
<tr>
<td>len</td>
<td>Specifies the length of the data buffer.</td>
</tr>
</tbody>
</table>

The routine should return the number of bytes actually written.
5.3.1.7 Performing Control Functions

The calling syntax for the xxIoctl() routine is as follows:

```c
STATUS xxIoctl
{
    DEVICE *xxDEV,  /* address of device descriptor */
    int code,      /* function code */
    long arg       /* function-specific argument */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver's device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type DEV_HDR. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>code</td>
<td>Specifies the request code of the control function to be performed.</td>
</tr>
<tr>
<td>arg</td>
<td>Specifies an argument (if any) to the control function.</td>
</tr>
</tbody>
</table>

The routine should return OK if the operation was successful; otherwise, the routine should return ERROR.

5.3.2 Block Device Functions

Under VxWorks for Alpha, block devices have a different interface than other I/O devices. Instead of interacting directly with the I/O system, block device drivers interact with a file system. The file system, in turn, interacts with the I/O system. This arrangement allows a single block device driver to be used with different file systems and reduces the number of I/O functions that must be supported in the driver.

The following I/O functions are generally required for block device drivers:

- `xxBlkRd()` — read one or more blocks of data from the device
- `xxBlkWrt()` — write one or more blocks of data to the device
- `xxIoctl()` — perform a control function on the device
- `xxReset()` — reset the device
- `xxStatusChk()` — check the status of the device

Note that the `xxReset()` and `xxStatusChk()` can be omitted if these functions are nonoperative for the device.
Your driver’s I/O routines must conform to the calling syntax of each function, as shown in the following subsections. Beyond that, you are free to implement your driver in any way appropriate. Since these routines will run in the calling task’s context, your driver is free to use any facilities normally available to tasks, including I/O to other devices.

Your driver can also provide other I/O functions, as appropriate to the device.

Throughout the following subsections the programming examples are taken from the driver for the NSC PC87311/PC87312 floppy disk controller. Note that within the VxWorks for Alpha I/O system, each driver has a short, unique name (such as “net” or “tyCo”) that is prefixed to each of its routine names — the xx in the list above. The name of the example driver is “fdc.”

5.3.2.1 Reading Blocks from the Device

The calling syntax for the xxBlkRd() routine is as follows:

```c
STATUS xxBlkRd
{
    DEVICE *xxDEV, /* address of device descriptor */
    int startBlk, /* starting block to read */
    int numBlks, /* number of blocks to read */
    char *pBuf /* pointer to data buffer */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver’s device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type BLK_DEV. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>startBlk</td>
<td>Specifies the number of the first block to read. Block numbers begin at zero. Any offset value used for this device must be added in by the driver.</td>
</tr>
<tr>
<td>numBlks</td>
<td>Specifies the number of blocks to read. The driver must emulate multiple-block reads if this feature is not supported by the hardware.</td>
</tr>
<tr>
<td>pBuf</td>
<td>Specifies a pointer to the buffer to receive the data read.</td>
</tr>
</tbody>
</table>

The routine should return OK if the transfer was successful; otherwise, it should return ERROR.
Example 5–2 shows a sample block read routine. Note that routine `fdcIo()`, called by this routine to perform the actual transfer, is shown in Example 5–10 in Section 5.8.

Example 5–2  Block Device Read Routine

```c
/*************************************************************************
** fdcBlkRd --
** This routine takes a pointer to the device data block, a starting block
** number, the total number of blocks and a pointer to the data buffer.
** It will select and spinup the drive and loop until all read block
** requests have been issued.
**
** ARGUMENTS: FDC_DEV *pdev ; Pointer to device data block
** int startblk ; Starting block number
** int numBlks ; Total number of blocks to write
** char *buffer ; Pointer to the data buffer
**
** RETURNS: STATUS
**
*/
STATUS fdcBlkRd(FDC_DEV *pdev, int startBlk, int numBlks, char *buffer)
{
    int status;
    int i;

    semTake (fdcMuteSem, WAIT_FOREVER);

    (continued on next page)```
Example 5–2 (Cont.) Block Device Read Routine

```c
fdcSpinup(pdev->fdc_sc.unit);
for (i=0; i<numBlks; i++) {
    status = fdcIo(pdev, buffer, startBlk, FDTOMEM);
    if (status != OK) {
        fdcSpindown(pdev->fdc_sc.unit);
        semGive(fdcMuteSem);
        return(ERROR);
    }
    buffer += pdev->blk_dev.bd_bytesPerBlk;
    startBlk++;
}
fdcSpindown(pdev->fdc_sc.unit);
semGive(fdcMuteSem);
return(OK);
```

5.3.2.2 Writing Blocks to the Device

The calling syntax for the `xxBlkWrt()` routine is as follows:

```c
STATUS xxBlkWrt(
    DEVICE *xxDEV, /* address of device descriptor */
    int startBlk, /* starting block to write */
    int numBlks, /* number of blocks to write */
    char *pBuf /* pointer to data buffer */
)
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver's device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type BLK_DEV. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>startBlk</td>
<td>Specifies the number of the first block to be written. Block numbers begin at zero. Any offset value used for this device must be added in by the driver.</td>
</tr>
<tr>
<td>numBlks</td>
<td>Specifies the number of blocks to write. The driver must emulate multiple-block writes if this feature is not supported by the hardware.</td>
</tr>
<tr>
<td>pBuf</td>
<td>Specifies a pointer to the buffer from which the data is to be written.</td>
</tr>
</tbody>
</table>

The routine should return OK if the transfer was successful; otherwise, it should return ERROR.
Example 5–3 shows a sample block write routine. Note that routine fdcIo(), called by this routine to perform the actual transfer, is shown in Example 5–10 in Section 5.8.

Example 5–3 Block Device Write Routine

```c
/**
 ** fdcBlkWr --
 **
 ** This routine takes a pointer to the device data block, a starting block
 ** number, the total number of blocks and a pointer to the data buffer.
 ** It will select and spinup the drive and loop until all write block
 ** requests have been issued.
 **
 ** ARGUMENTS: FDC_DEV *pdev ; pointer to the device data block
 ** int startblk ; Starting block number
 ** int numBlks ; Total number of blocks to write
 ** char *buffer ; Pointer to the data buffer
 **
 ** RETURNS: STATUS
 **
 ** /
 */

STATUS fdcBlkWr(FDC_DEV *pdev, int startBlk, int numBlks, char *buffer)
{
    int status;
    int i;

    semTake (fdcMuteSem, WAIT_FOREVER);

    fdcSpinup(pdev->fdc_sc.unit);
    for (i=0; i<numBlks; i++) {
        status = fdcIo(pdev, buffer, startBlk, MEMTOFD);
        if (status != OK) {
            fdcSpindown(pdev->fdc_sc.unit);
            semGive(fdcMuteSem);
            return(ERROR);
        }
    }

    return(OK);
}
```

(continued on next page)
Example 5–3 (Cont.)  Block Device Write Routine

    buffer += pdev->blk_dev.bd_bytesPerBlk;
    startBlk++;
}
fdcSpindown(pdev->fdc_sc.unit);
semGive(fdcMuteSem);
return(OK);
}

5.3.2.3 Performing Control Functions

The calling syntax for the `xxIoctl()` routine is as follows:

```c
STATUS xxIoctl(
    DEVICE *xxDEV, /* address of device descriptor */
    int code,    /* function code */
    long arg     /* function-specific argument */
)
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver's device descriptor structure. Note that the <code>typedef DEVICE</code> specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type <code>BLK_DEV</code>. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
<tr>
<td>code</td>
<td>Specifies the request code of the control function to be performed.</td>
</tr>
<tr>
<td>arg</td>
<td>Specifies an argument (if any) to the control function.</td>
</tr>
</tbody>
</table>

The routine should return OK if the operation was successful; otherwise, the routine should return ERROR. If the routine is called with an unknown code, the routine should set `errno` to `S_iolib_UNKNOWN_REQUEST` and return ERROR.

Example 5–4 shows a sample I/O control routine.
Example 5–4  Block Device I/O Control Routine

>Status fdcIoctl(FDC_DEV *pdev, int function, void *arg) {
  FAST int status;
  FAST int track;
  FAST int head;
  int retryCount;
  int type = pDev->fdc_sc.fdType;
  int unit = pDev->fdc_sc.unit;

  semTake (fdcMuteSem, WAIT_FOREVER);

  switch (function) {
    case FIODISKFORMAT:
      fdcSpinup (unit);
      (void) fdcRecal (unit);
  }

  return status;

}
Example 5–4 (Cont.) Block Device I/O Control Routine

```c
for (track = 0; track < fdTypes[type]->tracks; track++)
    for (head = 0; head < fdTypes[type]->heads; head++)
    {
        retryCount = 0;
        while (fdcSeek(unit, track, head) != OK)
            if (++retryCount > MAX_RETRY)
                (void) errnoSet (S_ioLib DEVICE_ERROR);
                status = ERROR;
                goto doneIoctl;
        retryCount = 0;
        while (fdcFormat(pDev, unit, track, head) != OK)
            if (++retryCount > MAX_RETRY)
                (void) errnoSet (S_ioLib DEVICE_ERROR);
                status = ERROR;
                goto doneIoctl;
    }
    status = OK;
    break;
default:
    errnoSet(S_ioLib UNKNOWN_REQUEST);
    status = ERROR;
    break;
}
doneIoctl:
    semGive (fdcMuteSem);
    fdcSpindown();
    return (status);
```
5.3.2.4 Resetting the Device

The calling syntax for the xxReset() routine is as follows:

```c
STATUS xxReset
{
    DEVICE *xxDEV /* address of device descriptor */
}
```

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver’s device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type BLK_DEV. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
</tbody>
</table>

This routine is called when the device is first mounted and should reset the device and controller. The routine should return OK if the operation was successful; otherwise, the routine should return ERROR.

If no reset function is required for the device, this routine can be omitted. In this case, the bd_reset field in the BLK_DEV structure should be set to NULL when the device is created.

Example 5–5 shows a sample reset routine.

**Example 5–5 Block Device Reset Routine**

```c
/**************************************************************************
* fdReset - reset a floppy disk controller
* This routine resets a floppy disk controller by doing a recalibrate.
* ARGUMENTS: FDC_DEV *pdev ; pointer to device data structure
* RETURNS: OK, always.
*/
STATUS fdcReset(FDC_DEV *pdev)
{
    fdcSpinup(pdev->fdc_sc.unit);
    fdcRecal(pdev->fdc_sc.unit);
    fdcSpindown();
}
(continued on next page)```
Example 5–5 (Cont.)   Block Device Reset Routine

    return(OK);

}  

5.3.2.5 Checking Device Status

The calling syntax for the xxStatusChk( ) routine is as follows:

    STATUS xxStatusChk
    {
        DEVICE *xxDEV /* address of device descriptor */
    }

where the routine parameters have the following meanings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxDEV</td>
<td>Specifies a pointer to the driver’s device descriptor structure. Note that the typedef DEVICE specifies a generic structure, as defined by the driver. The only requirement of this generic structure is that it begin with a structure of type BLK_DEV. (See Section 5.2 for a description of the device descriptor.)</td>
</tr>
</tbody>
</table>

This routine, if present, is called each time the file system receives an open( ) or creat( ) for the device. The routine should return OK if the open or create operation can continue. (If a new disk is present, the routine should set the bd_readyChanged field in the BLK_DEV structure to TRUE before returning OK.) Otherwise, the routine should set errno to a value indicating the problem and return ERROR.

If no status checking function is required for the device, this routine can be omitted. In this case, the bd_statusChk field in the BLK_DEV structure should be set to NULL when the device is created.

Example 5–6 shows a sample status check routine.
Example 5–6  Block Device Status Check Routine

******************************************************************************
* fdcStatusChk - Check the status of the floppy disk controller
*     The routine reads the MSR register and then checks the command in
* progress bit or drive busy. It also reads the SIO_DIR to check
* for the disk changed bit. If the disk changed bit is set then it
* sets the bd_readyChanged field in the BLK_DEV structure.
*     ARGUMENTS: FDC_DEV *pdev ; pointer to device data structure
*     RETURNS: STATUS
* /

STATUS fdcStatusChk(FDC_DEV *pdev)
{
    u_long flag,adrReg;
    int temp;
    flag = READ_BYTE;
    adrReg = SIO_DIR;
    FDC_READ(adrReg,flag,temp);
    if ((temp & FDC_DIR_DCH) != 0)
        pdev->blk_dev.bd_readyChanged = TRUE;
    adrReg = SIO_MSR;
    FDC_READ(adrReg,flag,temp);
    if (temp & ((1<<(pdev->fdc_sc.unit)) | FDC_MSR_CIP))
        errnoSet(EBUSY);
        return(ERROR);
    return(OK);
}

5.4 Implementing select()

Supporting select() in your driver allows calling tasks to wait for input from
multiple devices or to specify a maximum time to wait for the device to become
ready for I/O. You may want your driver to support select() if any of the
following is true:

- Calling tasks may want to specify a time to wait for I/O from the device
  (for example, a task may want to time out on a UDP socket if the packet
  never arrives).
• The driver supports multiple devices and a calling task may want to simultaneously wait for any number of them (for example, multiple pipes used for different data priorities).

• A calling task may want to wait for I/O from the device while also waiting for I/O from another device (for example, a server task using both pipes and sockets).

Basically, to support the select() function, your driver must keep a list of tasks waiting for device activity. When the device becomes ready, the driver unblocks all the tasks waiting for the device. Implementing this function is quite straightforward since most of the functionality is provided in selectLib.

First, your driver must declare a SEL_WAKEUP_LIST structure (typically as part of the device descriptor, see Section 5.2) and initialize it by calling selWakeupListInit(). This is done in the driver's device creation routine (see Section 5.9).

Then, when a task calls select(), selectLib calls the driver's xxIoctl() routine with a command of either FIOSELECT or FIOUNSELECT. If xxIoctl() is called with FIOSELECT, the driver must do the following:

1. Add the SEL_WAKEUP_NODE (provided as the function-specific argument of the call, arg) to the SEL_WAKEUP_LIST by calling selNodeAdd().

2. Call selWakeupType() to determine what the task is waiting for — data to be read from the device (SELREAD) or the device to be ready to be written (SELWRITE).

3. If the device is ready, as determined by the results of routine selWakeupType(), call selWakeup() to ensure that the task does not block.

If xxIoctl() is called with a code of FIOUNSELECT, the driver should call selNodeDelete() to remove the specified SEL_WAKEUP_NODE from the wakeup list.

When the device becomes available, the driver should call selWakeupAll() to unblock all tasks waiting for the device. This is typically done in the driver's interrupt service routine, but can be done anywhere that is appropriate. For example, a pipe driver might call selWakeupAll() from its xxRead() routine to unblock all tasks waiting to write since there is now room in the pipe to store the data. Similarly, the pipe driver's xxWrite() routine might call selWakeup() to unblock all tasks waiting to read.
Example 5–7 shows an example of driver code fragment supporting `select()`.
In this example, the driver unblocks tasks waiting for the device in its interrupt service routine.

Example 5–7 Driver Support for `select()`

```c
/* arkLib.h - header file for ark driver

typedef struct /* ARK_DEV */
{
    DEV_HDR devHdr; /* ark device header */
    BOOL arkDataAvail; /* read data available? */
    BOOL arkRdyWrite; /* device ready to write? */
    SEL_WAKEUP_LIST selWakeupList; /* list of waiting tasks */
} ARK_DEV;

/* arkDrv.c - code fragments for supporting select() */
#include "vxWorks.h"
#include "selectLib.h"

STATUS arkDevCreate
{
    char* name, /* name of ark to create */
    int num, /* number of arks to create */
    int aCnt /* number of animals living on ark */
}
{
    ARK_DEV* pArkDev; /* address of ark device descriptor */
    . . .
    /* allocate memory for ARK_DEV */
pArkDev = (ARK_DEV*) malloc((unsigned) sizeof (ARK_DEV + num * aCnt));
    . . .
    /* initialize wakeup list */
    selWakeupListInit (pArkDev->selWakeupList);
    . . .
}

STATUS arkIoctl
{
    ARK_DEV* pArkDev, /* address of ark device descriptor */
    int code, /* function code */
    int* arg /* function-specific code */
}
{
    . . .
```

(continued on next page)
Example 5–7 (Cont.)  Driver Support for select()

```c
switch (code)
{
  ...
  case FIOSELECT:
    /* add node to wakeup list */
    selNodeAdd (&pArkDev->selWakeupList, (SEL_WAKEUP_NODE*) arg);
    if (selWakeupType ((SEL_WAKEUP_NODE*) arg) == SELREAD
        && pArkDev->arkDataAvail)
      /* data available, make sure task does not block */
      selWakeup ((SEL_WAKEUP_NODE*) arg);
    if (selWakeupType ((SEL_WAKEUP_NODE*) arg) == SELWRITE
        && pArkDev->arkRdyWrite)
      /* device ready to write, make sure task does not block */
      selWakeup ((SEL_WAKEUP_NODE*) arg);
  case FIONUNSELECT:
    /* delete node from wakeup list */
    selNodeDelete (&pArkDev->selWakeupList, (SEL_WAKEUP_NODE*) arg);
    ...
}
void arkISR
(
  ARK_DEV *pArkDev
)
{
  ...
  /* if data available to read, wake up all waiting tasks */
  if (pArkDev->arkDataAvail)
    selWakeupAll (&pArkDev->selWakeupList, SELREAD);
  /* if device ready to write, wake up all waiting tasks */
  if (pArkDev->arkRdyWrite)
    selWakeupAll (&pArkDev->selWakeupList, SELWRITE);
}
```

5.5 Mapping the Bus

Alpha targets support bus mapping that is dynamically configurable by the software and consists of many separate mappings of smaller blocks. This type of mapping is known as scatter/gather mapping. The VMEbus uses scatter/gather mapping. Alpha targets can also have directly mapped buses,
such as the PCI bus on the AXPvme platform and the ISA bus on the EB64 platform.

You can use adpLib library routines to handle both types of mapping. Use adpCreateMapPath() and adpFreeMapPath() to create and free a dynamic mapping. For directly mapped buses, use adpCreateMapPath() to obtain the map base and size.

5.5.1 Mapping Base, Granularity, and Offset

Routine adpCreateMapPath() returns the base address (srcAddress) of either an allocated scatter/gather region or a directly mapped region. srcAddress always points to the start of the mapped region and may or may not point to the address requested (dstAddress). That is, srcAddress is the requested address (dstAddress) rounded down to the nearest mapping page boundary.

When you call the adpLib read and write routines — adpRead(), adpWrite(), adpMemRead(), and adpMemWrite() — you must supply the map base returned from adpCreateMapPath(), along with an offset to the desired address within the mapped region. You calculate this offset using the map granularity returned by adpCreateMapPath() (granularity). The formula is as follows:

\[
\text{offset} = \text{dstAddress} \& (\text{granularity} - 1);
\]

Scatter/gather maps must be naturally aligned according to the map granularity of the destination bus. On AXPvme platforms, the minimum map granularity for the VMEbus is 256 KBytes.

See the description of routine adpCreatMapPath() in the VxWorks Reference Manual for more information about mapped regions.

5.5.2 Pointer I/O

The adpLib read and write routines offer the simplest, most portable method for doing I/O. However, this can be at the expense of extra CPU cycles to perform address calculations, check arguments, and so forth. If it is important for your driver to have the fastest access speed possible, you may choose to perform your own address calculations and do I/O by dereferencing pointers to those locations in I/O space.

Pointer I/O can be done for both dense-space and sparse-space addresses, though the address calculation is more error prone in sparse space. See Chapter 3 for a description of dense-space and sparse-space addressing. Also see the VxWorks Programmer’s Guide for a detailed description of address calculation, as well as other programming considerations when doing direct pointer I/O.
5.6 Interrupt Handling

In addition to the routines necessary to process I/O requests from a calling task or file system, your driver needs one or more interrupt-level routines to process device interrupts.

5.6.1 Writing an Interrupt Service Routine

To process a device interrupt you write an interrupt service routine (ISR) for your driver. Example 5–8 shows the interrupt service routine for the NSC PC87311/PC87312 floppy disk controller.

Example 5–8 Device Driver Interrupt Service Routine

```c
/*********************************************************************
* fdcISR()
* This is the interrupt service routine for the driver. It gives
* back the binary semaphore to synchronize the I/O completion.
* ARGUMENTS: FDC_DEV *pdev ; pointer to device data structure,
* ; not currently used.
* RETURNS: N/A
*/
fdcISR(FDC_DEV *pdev)
{
    semGive(fdcBinSem);
}
```

5.6.1.1 Special Limitations of Interrupt Code

Many VxWorks facilities are available to an ISR, but there are some important limitations.

Under VxWorks, interrupt service code runs in a special context, outside the context of other tasks. Since no context switch is involved, very fast response time is possible. However, this also means an ISR has no task control block and all ISRs share a common stack. Thus, an ISR must not invoke any function that might cause it to block. This restriction has the following implications:

- An ISR must not try to take a semaphore. It can, however, give a semaphore on which tasks may be waiting.
• Memory routines `malloc()` and `free()` take a semaphore. Therefore, an ISR must not invoke these routines, or any routines that call them. For example, all creation and deletion routines are restricted from interrupt-level use.

• An ISR must not perform I/O through another VxWorks driver. Most device drivers require a task context since they typically block waiting for the device.

An important exception to this is the VxWorks pipe driver, which has been specifically designed to permit writes by interrupt service code. Also, the `logLib` subroutine library provides a system console text logging facility designed to be callable by ISRs.

• When control transfers into an ISR, the floating-point registers have not been saved. Therefore, if your ISR requires floating-point instructions, you must use the routines in `fppALib` to explicitly save and restore the floating-point coprocessor registers.

Refer to the VxWorks Programmer’s Guide for a complete list of routines that are callable by interrupt service code.

5.6.1.2 Exception Conditions at Interrupt Level

When a task causes a hardware exception (such as an illegal instruction or a bus error), the task is suspended and the rest of the system continues uninterrupted. However, when an ISR causes such an exception, it has no context that can be suspended. Instead, the VxWorks software stores a description of the exception in low memory and executes a system restart. The boot ROM prints the exception description on the system console.

5.6.2 Connecting to Interrupts

Your driver’s ISR must be “connected” to the corresponding device interrupt vector. You do this by calling routine `adpIntConnect()` from the driver initialization routine.

Example 5-9 shows this call for the NSC PC87311/PC87312 floppy disk driver.
Example 5–9  Connecting to an Interrupt

/* connect up the ISR */

status = adpIntConnect(ISA_VECTOR(0, ISA_INT_FLOPPY_DISK),
                        0, /* ISR parameter */
                        fdcISR, /* ISR address */
                        fdcReset, /* reset routine */
                        0); /* power fail routine */

5.7 Communication and Synchronization

When a user invokes one of the basic I/O functions, requiring service of a
device driver, the VxWorks I/O system routes the request to the appropriate
routine of the driver. The driver's routine then runs in the context of the
calling task, just as if it had been called directly. Thus, the driver is free to use
any of the facilities normally available to tasks.

Drivers typically communicate with their calling tasks through shared data
structures.

The usual mechanism for synchronization between a driver and its calling task
(or other drivers) is the semaphore facility provided in the semLib library.

5.8 DMA Transfers

To implement support for DMA transfers, your driver must first set up an
inbound mapping from the device's bus adapter to the primary bus adapter.
You use routine adpCreateMapPath() to create this mapping. (See Section 5.5.)

Routine adpCreateMapPath() creates the hardware data path in one of two
ways:

• It allocates scatter/gather maps for bus bridges with scatter/gather
capabilities.

• It determines the starting address of a directly mapped bus bridge, across
as many bus bridges as it takes to make an end-to-end mapping.

After allocating and loading all mapping registers along the data path,
the routine returns the source address, the map object handle, the real
size of the map, and the primary bus granularity. (See the description of
adpCreateMapPath() in the VxWorks Reference Manual for more information
about the use of this routine.)

To perform the actual DMA transfer, your driver must call the bus-specific
routine provided in the board support package of your system. See the
hardware supplement for your target platform for details of this routine.
Example 5–10 shows the fdcIo() routine of the NSC PC87311/PC87312 floppy disk driver and illustrates the processing necessary for a DMA transfer.

Example 5–10  DMA Processing

`**********************************************************************************
**
** fdcIo --
**
** This routine takes a pointer to the device data block, a pointer to
** the data buffer, a block or sector number and a read/write flag.
** It sets up the dma engine and transfers one block of data to or
** from the disk.
**
** ARGUMENTS:  FDC_DEV *pdev ; pointer to the device data block
** char  *buf ; Pointer to the data buffer
** int sec ; Starting block/sector number
** int type ; Flag distinguishing a read or write
**
** RETURNS:  STATUS
**
**
**********************************************************************************`

```c
int fdcIo(pdev, buf, sec, type)
    FDC_DEV *pdev;
    char  *buf;
    int sec;
    int type;
{
    int dhead;
    int dtrk;
    int dsec;
    char cmd[MAX_CMD];
    char results[7];
    int flag,size,status;
    volatile u_int *tmp;
    u_long access,adr_space;
    int numSec = 1;
    int fdType = pdev->fdc_sc.fdtype;
    /*
    ** these are needed for the sysIsaDma call
    */
    access = ACCESS_TYPE_BYTE;
    adr_space = ADDRESS_SPACE_MEM;
    size = numSec * (128 << fdTypes[fdType]->M_secSize);
    (continued on next page)
```
Example 5–10 (Cont.) DMA Processing

```c
if (fdcctrk < 0) {
    if (fdcRecal(pdev->fdc_sc.unit) == ERROR) {
        return(ERROR);
    }
    fdcctrk = 0;
}

dhead = 0;
dtrk = sec / (fdTypes[fdType]->secPerTrk * 
    fdTypes[fdType]->heads);
dsec = sec % (fdTypes[fdType]->secPerTrk * 
    fdTypes[fdType]->heads);
if (dsec >= fdTypes[fdType]->secPerTrk) {
    ++dhead;
    dsec -= (fdTypes[fdType]->secPerTrk);
}
if (dtrk != fdcctrk) {
    status = fdcSeek(pdev->fdc_sc.unit,dtrk,dhead);
    if (status != OK){
        return(ERRGR);  
    }
    fdcctrk = dtrk;
}
/* control */
/* set up the S10 DMA control data structure to be passed to sysIsaDma 
** for the actual write. */
```
```
pDmaOpt->dma1_cmd_mode = 0x0;  /* Take the default command mode*/
pDmaOpt->dma2_cmd_mode = 0x0;
pDmaOpt->dma1_xtend_mode = S10_DCM_CH2;
pDmaOpt->dma2_xtend_mode = S10_DCM_CH2;
```

(continued on next page)
Example 5–10 (Cont.) DMA Processing

    if (type == FDTOMEM) { /* if read from disk */
        pDmaOpt->dma1_chan_mode = (SIO_DCM_SINGLE | SIO_DCM_WRITE |
            SIO_DCM_CH2);
        pDmaOpt->dma2_chan_mode = SIO_DCM_CASCADE; /* Take default of verify
            xfer */
    } else {
        pDmaOpt->dma1_chan_mode = (SIO_DCM_SINGLE | SIO_DCM_READ |
            SIO_DCM_CH2);
        pDmaOpt->dma2_chan_mode = SIO_DCM_CASCADE; /* Take default of verify
            xfer */
    }
    pDmaOpt->channel = 0x2; /* floppy uses channel 2*/
    /* unmask the appropriate channels to enable and start the dma transfer */
    /* always enable channel 4 for the cascade */
    pDmaOpt->dma1_ena_chan = SIO_DCM_CH2;
    pDmaOpt->dma2_ena_chan = SIO_DCM_CH4;

    status = sysIsaDma(buf, NULL, adr_space, access, size, pDmaOpt);
    if (status != OK){
        return(ERROR);
    }
    /* read or write */
    cmd[0] = (type==FDTOMEM) ? FDC_CMD_RD : FDC_CMD_WR;
    cmd[1] = (dhead<<2)|(pdev->fdc_sc.unit); /* head # or’d w/unit #*/
    cmd[2] = dtrak; /* track or cylinder */
    cmd[3] = dhead; /* head */
    cmd[4] = dsec+1; /* sector */
    cmd[5] = fdTypes[fdType]->M_secSize; /* bytes per sector */
    cmd[6] = fdTypes[fdType]->secPerTrk; /* sectors per track */
    cmd[7] = fdTypes[fdType]->gap1; /* inter sec gap size*/
    cmd[8] = 0xFF; /* recommended value */
    fdcCmd(cmd, FDC_CMD_LEN_RD); /* rd, wr lgth the same */

    semTake (fdcBinSem, WAIT_FOREVER);

    flag = fdcResultPhase(0x7,&results);
    if ( (flag == 0) && ((results[0] & 0xC0 == 0x00) ){
        return(OK);
    }

    /* the make file system call initializes without dma the first two
    ** tracks of the disk, and the expected results are an abnormal
    ** termination with the EOT bit set in ST!
*/

(continued on next page)
Example 5–10 (Cont.) DMA Processing

```c
else if ( ((results[0] & 0xC0) == FDC_ST0_ERR) &&
            ((results[1] & FDC_ST1_ET) == FDC_ST1_ET) )
    return(OK);

else if ((results[1] & FDC_ST1_WP) == FDC_ST1_WP){
    (void) errnoSet (S_ioLib_WRITE_PROTECTED); return(ERROR);
}

else if ( ((results[1] & 0xFD) != 0) || ((results[2] & 0xFF) != 0)){
    (void) errnoSet (S_ioLib_DEVICE_ERROR); return(ERROR);
}

return(ERROR);

```

5.9 Driver and Device Initialization Routines

In addition to the routines that implement I/O and control functions, device drivers also typically have two other routines:

- A routine that performs driver initialization
- A routine that identifies and initializes the devices to be serviced by the driver

These routines are generally called at system startup from the system initialization task, `usrRoot()`, although there may be circumstances under which you may choose to invoke them at another time. See Section 5.11 for a description of the procedure for loading and configuring your driver.

As was the case with I/O and control functions, these driver and device initialization routines differ for block and nonblock devices.

5.9.1 Nonblock Initialization Routines

The driver and device initialization routines for nonblock device drivers interface directly with the I/O system.

5.9.1.1 Installing the Driver

The primary responsibility of the driver initialization routine for a nonblock device is to install the driver in the I/O system driver table. The driver table contains an array of addresses of the routines to be called in response to each of the seven basic I/O functions.
To install the driver, your driver’s initialization routine (typically named xxDrv(), where xx is the generic name of the driver) calls the system routine iosDrvInstall(). The arguments to this routine are the addresses of the seven I/O routines. (Nonoperative routines are specified with a NULL address.) After entering the addresses in a free slot in the driver table, isoDrvInstall() returns the driver table index. This index is known as the driver number and is subsequently used to associate individual devices with the driver (see Section 5.9.1.2).

Routine xxDrv() also performs any other operations that are done only once for the driver (as opposed to any device-specific initialization), such as connecting to the interrupt service routines and allocating any necessary data structures.

Example 5–11 shows abbreviated code for the driver initialization routine of a hypothetical nonblock device driver named “nblk.”

**Example 5–11 Nonblock Device Driver Initialization Routine**

```c
/************************************************************
* nblkDrv - driver initialization routine
*
* This routine is called to initialize the driver. It installs the
* driver by calling iosDrvInstall. It may also allocate and init data
* structures, connect interrupt routines to their interrupts, and init
* device hardware.
*/
STATUS nblkDrv ()
{

  nblkDrvNum = iosDrvInstall (0, 0, nblkOpen, 0, nblkRead,
                         nblkWrite, nblkIoctl);

  ...

  (void) adpIntConnect (nblkVec, 0, nblkISR, . . . );

  ...
}
```

### 5.9.1.2 Identifying Devices

In addition to a driver initialization routine, your driver must include a routine that identifies the devices it will service and adds them to the I/O system. This routine is typically named xxDevCreate(), where xx is the generic name of the driver.

Nonblock devices are added to the I/O system by calling system routine iosDevAdd(). The arguments to this routine are the address of the device descriptor (see Section 5.2), the device’s name, and the driver number returned from iosDrvInstall() (see Section 5.9.1.1). Routine iosDevAdd() loads the
device name and driver number into the device header in the descriptor and then adds the address of the device descriptor to the system device list.

Routine xxDevCreate() also generally performs other device-specific operations required to initialize the device.

Example 5–12 shows abbreviated code for the device initialization routine of a hypothetical nonblock device driver named “nblk.”

Example 5–12 Nonblock Device Initialization Routine

```c
/*****************************************************************
* nblkDevCreate - device initialization routine
*
* This routine is called to add a device serviced by the driver to
* the system. The only required argument of this routine is the
* device name. Other driver-dependent arguments may be required,
* such as buffer sizes, device addresses, and so on.
* The routine adds the device to the I/O system by calling iosDevAdd.
* It may also allocate and initialize data structures for the device,
* initialize semaphores, initialize the device hardware, and so on.
*/
STATUS nblkDevCreate(name, . . . )
   char *name;
   . . .
   {
      status = iosDevAdd(nblkDev, name, nblkDrvNum);
      . . .
   }
```

5.9.2 Block Initialization Routines

Since block device drivers do not interface directly with the I/O system, they are not installed in the system driver list. Instead, the associated file system is installed as the driver. Each file system has only one entry in the driver table, even though several different device drivers may have devices served by that file system.

After a device is initialized for use with a particular file system, all I/O operations for the device are routed through that file system. The file system, in turn, routes specific device operations to the routines whose addresses are defined in the device descriptor (see Section 5.2).
5.9.2.1 Initializing the Driver

Although it is not specifically required by the VxWorks software, block device drivers normally have a general initialization routine that performs all startup operations for the driver. This routine is typically named \texttt{xxInit()}, where \texttt{xx} is the generic name of the driver.

The functions performed by this routine are generally those that affect the device controller, rather than the specific devices on that controller. Common operations in a block device driver initialization routine include the following:

- Hardware initialization
- Allocation and initialization of common data structures
- Semaphore creation
- Interrupt vector initialization
- Interrupt enabling

As noted above, block device drivers do not install themselves in the system driver table.

Example 5–13 shows the driver initialization routine for the NSC PC87311/PC87312 floppy disk controller driver.

Example 5–13  Block Device Driver Initialization Routine

```c
/********************************************************************
* fdcInit()
* This is the driver initialization routine. This routine should
* perform all operations which are done once only, as opposed to
* functions which must be performed for each device served by the
* driver. For example, hardware initialization, data structure
* allocation and initialization, and so on.
* Unlike a nonblock device driver, this routine does not call
* iosDrvInstall(). This driver does not appear in the driver tables,
* but in the file system tables. All I/O done by this driver will be
* done as requests from the file system.
* ARGUMENTS: N/A
* RETURNS: STATUS
*/
```

(continued on next page)
Example 5–13 (Cont.)  Block Device Driver Initialization Routine

FD_TYPE *fdTypes[NUM_FDTYPES];  /* floppy drive tables */
DMA_OPT *pDmaOpt;  /* DMA options structure */
SEM_ID fdc_mutex;  /* mutex to synchronize access */
/* to driver global data */
SEM_ID fdcMuteSem;  /* mutex semaphore */
SEM_ID fdcBinSem;  /* bin semaphore for synch */

STATUS fdcInit()
{
    int unit, i;
    int status;

    fdc_mutex = semBCreate(SEM_Q_FIFO, SEM_FULL);
    if (fdc_mutex == (SEM_ID) NULL) {
        return(ERROR);
    }

    fdcMuteSem = semMCreate(SEM_Q_PRIORITY | SEM_DELETE_SAFE |
    SEM_INVERSION_SAFE);
    if (fdcMuteSem == (SEM_ID) NULL) {
        return(ERROR);
    }

    fdcBinSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
    if (fdcBinSem == (SEM_ID) NULL) {
        return(ERROR);
    }

    /* create the dma options data structure */
    pDmaOpt = (DMA_OPT *) malloc ((unsigned) sizeof (DMA_OPT));
    if (pDmaOpt == NULL){
        return(ERROR);
    }

    /* Set up the floppy drive table, currently only one drive type
    ** is supported, 3.5" 1.44M 2hd. */
    for (i =0; i < NUM_FDTYPES; i++){
        fdTypes[i] = (FD_TYPE *) calloc(1, sizeof(FD_TYPE));
    }
}

(continued on next page)
Example 5–13 (Cont.) Block Device Driver Initialization Routine

fdTypes[0]->secPerTrk = 18;
fdTypes[0]->tracks = 160;
fdTypes[0]->sectors = fdTypes[0]->secPerTrk * fdTypes[0]->tracks;
fdTypes[0]->secSize = 512; /* actual size required by filesys */
fdTypes[0]->M_secSize = 2; /* mask bit = 512 for rd/wr/format */
fdTypes[0]->heads = 2;
fdTypes[0]->gap1 = 0x1B;
fdTypes[0]->gap2 = 0x6C;
fdTypes[0]->dataRate = FDC_CCR_500;
fdTypes[0]->stepRate = FDC_CMD_SP_STP;
fdTypes[0]->mfm = 0; /* not used */
fdTypes[0]->sk = 0; /* not used */

/* connect up the ISR */

status = adpIntConnect(ISA_VECTOR(0,ISA_INT_FLOPPY_DISK),
                        0, /* ISR parameter */
                        fdcISR,   /* ISR address */
                        fdcReset, /* reset routine */
                        0);       /* power fail routine */

if (status == ERROR) {
    return(ERROR);
}

adapter = adpLookup(ISA_VECTOR(0,0));
if (adpSetBusIntEna(adapter, ISA_VECTOR(0,ISA_INT_FLOPPY_DISK),
                    ISA_INT_FLOPPY_DISK) != OK) {
    return(ERROR);
}

/* set up outbound mapping */

(continued on next page)
Example 5–13 (Cont.) Block Device Driver Initialization Routine

```c
if (adpCreateMapPath
    (NULL, /* src adapter (NULL = cpu) */
    adapter, /* dest adapter */
    0, /* dest address to map*/
    1, /* rqstd map size, NA for direct map */
    0, /* node number of requester */
    ADDRESS_SPACE_IO, /* addr space */
    ACCESS_TYPE_BYTE | ACCESS_TYPE_WORD, /* default access type */
    &fdcOSrcAddr, /* source bus address of map */
    &fdcOHandle, /* handle of created map object */
    &fdcORealSize, /* created map size */
    &fdcOGranularity /* dest bus mapping granularity */
) == ERROR) {
    #ifdef INCLUDE_LOGGING
        logMsg ("fdcInit: outbound adpCreateMapPath failed\n");
    #endif /* INCLUDE_LOGGING */
    return ERROR;
}

/* Create an inbound mapping from adapter to main memory for DMA */
if (adpCreateMapPath
    (adapter, /* src adapter */
    NULL, /* dest adapter (NULL = main mem) */
    0, /* dest address to map */
    1, /* requested size of map */
    0, /* node number, unused */
    0, /* default addr space */
    &fdcISrcAddr, /* source bus address of map */
    &fdcIHandle, /* handle of created map object */
    &fdcIRealSize, /* created map size */
    &fdcIGranularity /* dest bus mapping granularity */
) == ERROR) {
    #ifdef INCLUDE_LOGGING
        logMsg ("fdcInit: inbound adpCreateMapPath failed\n");
    #endif /* INCLUDE_LOGGING */
    return ERROR;
}

for (unit=0; unit<FDC_MAX_DRIVES; unit++) {
    fdc_drives[unit] = NULL;
}
if (fdcInitDrive(FDC_CCR_500))
    return(ERROR);
```

(continued on next page)
5.9.2.2 Initializing Devices

A block device driver must furnish a routine to create and initialize a logical disk device for each device it services. This routine is typically named \texttt{xxBlkCreate()}, where \texttt{xx} is the generic name of the driver.

The primary responsibility of \texttt{xxBlkCreate()} is to allocate and initialize the device descriptor (see Section 5.2). Unlike the device initialization routine required for nonblock device drivers, it does not call \texttt{iosDevAdd()} to install the device in the I/O system device table. Instead, the routine returns the address of the properly initialized device descriptor. This address is then passed to the device initialization routine of the appropriate file system — \texttt{dosFsDevInit()}, \texttt{rt11FsDevInit()}, or \texttt{rawFsDevInit()}. The file system routine then calls \texttt{iosDevAdd()} to register the device as one of its own.

In addition to allocating the device descriptor, \texttt{xxBlkCreate()} may also perform device-specific operations required to initialize the individual device.

Example 5–14 shows the device initialization routine for the NSC PC87311/PC87312 floppy disk controller driver.

Example 5–14 Block Device Initialization Routine

```c
/******************************************************************************
* fdcBlkCreate - device creation and initialization *
* Check to be sure device doesn’t already exist. If it doesn’t, *
* then create and initialize it. *
* *
* ARGUMENTS: char *name ; device name *
* int unit ; drive unit number *
* int type ; media type, currently *
* ; 3.5" 1.44Mb supported *
* *
* RETURNS: Pointer to device descriptor block, or NULL on error *
* *
*/
long fdcBlkCreate (char *name, int unit, int type)
```

(continued on next page)
Example 5–14 (Cont.) Block Device Initialization Routine

```
{  
  FDC_DEV *pFdcDev;
  DMA_OPT *dmaOpt;
  int i, status, tmp, memSize;
  u_long h, r, g;
  char *fdcOBuffer,*fdcIBuffer;
  char startchar, boink, endchar;
  int bytot, x;
  bytot = 1 * 1024;
  if ((unit < 0) || (unit > FDC_MAX_DRIVES)) {
    errnoSet (ENODEV);
    return(NULL);
  }
  /* first lock driver structures */
  semTake(fdcMuteSem, WAIT_FOREVER);
  /* search for device, perhaps it is already created */
  if (fdc_drives[unit] != (FDC_DEV *) NULL) {
    semGive(fdcMuteSem);
    return ((long)(fdc_drives[unit]));
  }
  /* OK, the device doesn’t exist yet, so create it */
  pFdcDev = (FDC_DEV *) (malloc ((unsigned) sizeof (FDC_DEV)));
  if (pFdcDev == NULL) {
    return(NULL);
  }
  /* create device mutex */
  pFdcDev->fdc_sc.mutex = semBCreate(SEM_Q_FIFO, SEM_FULL);
  if (pFdcDev->fdc_sc.mutex == (SEM_ID) NULL) {
    free(pFdcDev);
    semGive(fdcMuteSem);
    return(NULL);
  }
  /* take device mutex */
  semTake(pFdcDev->fdc_sc.mutex, WAIT_FOREVER);
  /* tell driver about this device */
  fdc_drives[unit] = pFdcDev;
  /* and release driver mutex */
  semGive(fdcMuteSem);
  /* Now set up the block device routines needed by the file system */
}
```
Example 5–14 (Cont.) Block Device Initialization Routine

```c
pFdcDev->blk_dev.bd_blkRd = fdcBlkRd;
pFdcDev->blk_dev.bd_blkWrt = fdcBlkWrt;
pFdcDev->blk_dev.bd_ioctl = fdcIoctl;
pFdcDev->blk_dev.bd_reset = fdcReset;
pFdcDev->blk_dev.bd_statusChk = fdcStatusChk;
pFdcDev->blk_dev.bd_removable = TRUE;
pFdcDev->blk_dev.bd_nBlocks = fdTypes[type]->sectors;
pFdcDev->blk_dev.bd_bytesPerBlk = fdTypes[type]->secSize;
pFdcDev->blk_dev.bd_bksPerTrack = fdTypes[type]->secPerTrk;
pFdcDev->blk_dev.bd_nHeads = fdTypes[type]->heads;;
pFdcDev->blk_dev.bd_retry = 3;
pFdcDev->blk_dev.bd_mode = UPDATE;
pFdcDev->blk_dev.bd_readyChanged = TRUE;
/* give up device mutex */
semGive(pFdcDev->fdc_sc.mutex);

/* finally, return the pointer to the device */
return((long) pFdcDev);
```

5.10 Bus-Specific Features

The adpLib I/O library provides routines that let you access bus-specific features. In particular, the following features are supported:

- Read-modify-write operation — Some buses, such as the VMEbus, support an atomic read and write operation through a read-modify-write cycle on the bus. Routine `adpRMW()` emulates this feature in the software.

- Posting, checking and clearing bus interrupts — In addition to sending interrupt requests to the CPU, some bus adapters can post interrupts for other devices to process. Routines `adpPostInterrupt()`, `adpCheckInterrupt()`, and `adpClearInterrupt()` support this feature.

- Resetting the bus or bridge adapter — The `adpReset()` routine resets and initializes the bus or bridge hardware.

Consult the Alpha architecture portion of the VxWorks Programmer’s Guide, and the hardware supplement for your particular target platform, for any bus- or platform-specific restrictions imposed on your device driver.
5.11 Loading and Configuring the Driver

When you finish programming your device driver, you compile it the same way you would compile any VxWorks application. (Refer to the VxWorks Programmer’s Guide for details.)

5.11.1 Loading the Driver Object

Once you have a compiled object module for your driver, there are several ways you can include it in the VxWorks for Alpha system.

5.11.1.1 Loading With the Shell

Perhaps the easiest way to include your driver object in the VxWorks system is to load it dynamically through the VxWorks shell once the system has been booted.

On your host system, set the default directory to the directory containing your driver object module and then invoke the VxWorks ld command, as follows:

```
% cd "/yourpath/yourdir"
% ld < yourobj
```

where /yourpath/yourdir is the directory path and yourobj is the name of your driver object module.

5.11.1.2 Loading With the Application Interface

You can also load your device driver dynamically from your application program using routine loadModule() from the loadLib library. To do this, include the following code sequence in your application:

```
long fd;
...
fd = open ("/yourpath/yourdir/yourobj", READ);
loadModule (fd, ALL_SYMBOLS);
close (fd);
```

where /yourpath/yourdir is the directory path and yourobj is the name of your driver object module.
5.11.1.3 Direct Linking

When your device driver is completely debugged, you can link it directly into the VxWorks system image. In fact, if your target system runs as a standalone system, this may be the only way to load your driver.

To link your driver into the VxWorks system image, you must modify the "makefile" in the board support package (BSP) for your target platform. The BSP is in yourCloneTree/config/target, where target is the name of your target system. For example, the BSP directory for an AXPvme target is yourCloneTree/config/AXPvme.

In the BSP directory, locate the makefile named Makefile.cputool where cputool is the designation for the target and host you are using. For example, the makefile for building an AXPvme target system on a Digital UNIX host is Makefile.21064osf.

Edit this makefile and locate the line defining either of the following symbols: MACH_EXTRA or MACH_EXTRA_ST. Add the path and name of your driver object module following any modules listed, separated by a single space.

MACH_EXTRA = obj1 obj2 /yourpath/yourdir/yourobj

or

MACH_EXTRA_ST = obj1 obj2 /yourpath/yourdir/yourobj

where /yourpath/yourdir is the directory path and yourobj is the name of your driver object module.

If you are building a system with a dynamically loaded symbol table, add your object module to the line defining MACH_EXTRA. If you are building a standalone system, add your object module to the line defining MACH_EXTRA_ST. If you are not sure which symbol to use, you can add your object module to both lines.

Once you have made this change, exit from the editor and rebuild the VxWorks system.

______________________________ Note ________________________________

The file Makefile.cputool is generated each time makefiles are rebuilt from makefile skeleton MakeSkel in the BSP directory. Any changes you make to Makefile.cputool will be lost if other configuration changes require a rebuild from MakeSkel. In this event, you will have to remodify the file.
5.11.2 Configuring the Driver

Before your driver can be accessed by an application, the driver and its devices must be configured into the VxWorks I/O system. If there is no need for your device to be active when the VxWorks system is first started up, the application can simply call the driver and device initialization routines (specified in Section 5.9) before issuing any I/O requests.

However, to have your driver and its devices configured at system startup, the driver's initialization routines can be called during the VxWorks system initialization sequence executed by routine `usrRoot()`. You can modify this routine to include calls to your driver's initialization routines.

The `usrRoot()` module is included in the file `yourCloneTree/config/all/usrConfig.c`. Edit `usrConfig.c` according to the instructions below.

---

**Note**

The I/O system must be initialized before you can configure your driver. Therefore, within `usrRoot()` look for a call to `iosInit()`. All calls to your driver's initialization routines must be placed after this call. Note also that other configuration details may require you to place the calls to initialize your driver at some point further down within `usrRoot()`.

---

- **Configuring a Nonblock Device Driver**

Insert a call to your driver initialization routine, followed by a call to the device initialization routine for each device to be serviced by the driver.

```c
status = xxDrv(); /* init the driver */
status = xxDevCreate(name, ...); /* init and add devices */
```

- **Configuring a Block Device Driver**

Before you can configure your driver, the appropriate file system must be initialized. Therefore, look for a call to the proper initialization routine — `dosFsInit()`, `rt11FsInit()`, or `rawFsInit()`. Routine `usrRoot()` calls these initialization routines based on the following conditional symbol definitions:

- **INCLUDE_DOSFS** — if defined, `usrRoot()` calls `dosFsInit()`
- **INCLUDE_RT11FS** — if defined, `usrRoot()` calls `rt11FsInit()`
- **INCLUDE_RAWFS** — if defined, `usrRoot()` calls `rawFsInit()`
To include the file system for your device, you must ensure that the appropriate symbol is defined in the `yourCloneTree/config/all/configAll.h` configuration file.

Following the call to `xxxFsInit()`, add a call to your driver’s initialization routine, if any.

```c
status = xxInit();
```

Follow this with the following pair of calls for each device serviced by the driver:

```c
/* create the device */
   devDesc = xxBlkCreate (devName, . . . );
/* register it with the file system */
   volDesc = xxxFsDevInit (devName, devDesc, . . . );
```

where `xxxFsDevInit()` is the device initialization routine for the appropriate file system — `dosFsDevInit()`, `rt11FsDevInit()`, or `rawFsDevInit()`.
Writing a Serial Device Driver

This chapter explains how to write a device driver for an onboard serial controller that is not already supported by VxWorks for Alpha. It describes the routine interfaces that connect a driver to the I/O system for access through regular I/O system interfaces such as open() and character I/O routines.

Before you undertake writing a serial driver, first determine if VxWorks already supports a driver for the serial controller on your target board. In addition to the per-BSP tyCoDrv source modules available in yourCloneTree/config/target, additional serial driver templates are available in yourCloneTree/src/drv/serial.

If your chip is supported, you do not need to write a new driver; rather, you modify the corresponding serial driver module as necessary to account for board dependencies, modify tyCoDrv initialization calls in module usrConfig.c to reflect your configuration, and place the tyCoDrv module in the BSP directory for your target.

If VxWorks does not support a driver for your onboard serial controller, you write a driver using this chapter and the serial driver templates supplied in the VxWorks tools and BSP kits.

Writing a serial device driver includes the following steps:

1. Select one or more serial driver templates from among those available in BSP target directories yourCloneTree/config/target/tyCoDrv.c and yourCloneTree/src/drv/serial.

2. Code the well-defined set of driver routines expected by the I/O system and initialization code, described in Section 6.3. As part of this, the driver must define device-specific data structures, described in Section 6.2.

3. Edit the tyCoDrv routine invocations in the root task, usrRoot(), in module yourCloneTree/config/all/usrConfig.c, to reflect your serial configuration.
4. Adjust the definitions of NUM_TTY and CONSOLE_TTY, the defaults for which are established in yourCloneTree/config/all/configAll.h, to reflect the number and numbering of serial devices created in your configuration. You can override the defaults in your target-specific configuration header file, yourCloneTree/config/target/config.h.

This chapter addresses the following topics:

- Driver overview, Section 6.1
- Driver data structures, Section 6.2
- Driver operations, Section 6.3
- Sample serial device driver, Section 6.4

6.1 Driver Overview

VxWorks serial I/O devices are buffered serial byte streams. Each device has a ring buffer for both input and output. Reading from a serial device, such as a terminal, extracts bytes from the input ring, and writing to a serial device adds bytes to the output ring. The size of the ring buffer is specified when the device is created, during system initialization.

Serial drivers adhere to the VxWorks I/O system structure, as described in the I/O System chapter of the VxWorks Programmer’s Guide. The routine tyCoDrv() is called once during initialization, usually from usrConfig.c, to initialize the driver; then tyCoDevCreate() is called to initialize each terminal (serial port). Once the driver and devices are initialized, the driver is accessed only through the I/O system.

A serial driver is expected to provide a well-defined set of device-specific routines, described in Section 6.3, for the higher levels of the I/O system to access.

The information needed for the I/O system to access a serial driver is provided in the driver’s TY_DEV structure, defined in tyLib.h and described in Section 6.2.

To implement the driver-specific functions expected by the I/O system, the driver can make use of the serial I/O support routines provided by the routine library tyLib. Table 6-1 lists those routines; for more information, see the description of tyLib in the VxWorks reference material.
Because the generic serial line library, tyLib, contains most of the code used by tyCoDrv, porting a console driver to a new board becomes a straightforward exercise. The tyCoDrv source you create in your BSP supplies only necessary hardware initialization functions, an interrupt service routine, and a start-output function. tyCoDrv() and tyCoDevCreate() function calls at system initialization hook up the board-specific routines to the I/O system and to tyLib.

Table 6–2 shows the initialization and runtime flow of control of the tyCoDrv serial driver.

### Table 6–2  Serial Driver Flow of Control

<table>
<thead>
<tr>
<th>Initialization</th>
<th>tyCoDrv.c (driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>usrRoot() → tyCoDrv()</td>
<td></td>
</tr>
<tr>
<td>Connects interrupts, installs the driver, and initializes the chip</td>
<td></td>
</tr>
<tr>
<td>→ intConnect()</td>
<td>intArchLib</td>
</tr>
<tr>
<td>→ iosDrvInstall()</td>
<td>iosLib</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 6–2 (Cont.) Serial Driver Flow of Control

**Initialization**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>usrRoot()</td>
<td>Initializes the channel, allocates buffers, initializes the device, and adds the device to the system</td>
<td>tyCoDrv.c (driver)</td>
</tr>
<tr>
<td>tyCoDevCreate()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tyDevInit()</td>
<td></td>
<td>tyLib</td>
</tr>
<tr>
<td>iosDevAdd()</td>
<td></td>
<td>iosLib</td>
</tr>
</tbody>
</table>

**Runtime**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O system</td>
<td>tyIoctl()</td>
<td>, other driver routines</td>
<td>tyCoDrv.c (driver)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>As necessary to handle board dependencies</td>
<td></td>
</tr>
<tr>
<td>Device</td>
<td>tyCoInt()</td>
<td>ISR</td>
<td>tyCoDrv.c (driver)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Handles device interrupts</td>
<td></td>
</tr>
</tbody>
</table>

### 6.2 Driver Data Structures

Each serial device driver defines a TY_CO_DEV structure, which contains driver specific information, including register information. The TY_CO_DEV structure contains another important driver data structure, TY_DEV, and is used to define an array, tyCoDv, for the number of channels on the device.

Example 6–1 shows the TY_CO_DEV structure defined in `yourCloneTree/h/drw/serial/z85c30.h` for all Z8C530 serial drivers, including the Digital AXPvme implementation, `yourCloneTree/config/AXPvme/tyCoDrv.c`, and the Motorola MVME147S-1 implementation.

**Example 6–1 Z8C530 Serial Driver Data Structure TY_CO_DEV**

```c
typedef struct /* TY_CO_DEV */
{
    TY_DEV tyDev;
    BOOL created; /* true if this device has really been created */
    unsigned int *cr; /* control register I/O address ycl change */
    unsigned int *dr; /* data port I/O address for flamingo */
    char intVec;    /* interrupt vector address ycl: unused, no change */
} TY_CO_DEV;
```

6–4 Writing a Serial Device Driver
The first field in each TY_CO_DEV structure is the TY_DEV structure, which is initialized by the tyDevInit() routine. Among the fields filled in at startup is a pointer to the driver's transmitter startup routine (tyCoStartup()). Example 6–2 shows the definition of TY_DEV, from yourCloneTree/h/tyLib.h. (Some related value and type definitions are omitted from this example, but are available in the header file.)

Example 6–2 Serial Driver Data Structure TY_DEV

```c
typedef struct /* TY_DEV - tty device descriptor */
{
    DEV_HDR devHdr; /* I/O device header */
    RING_ID rdBuf; /* ring buffer for read */
    SEMAPHORE rdSyncSem; /* reader synchronization semaphore */
    SEMAPHORE mutexSem; /* mutual exclusion semaphore */
    struct /* current state of the read channel */
    {
        unsigned char xoff; /* input has been XOFF’d */
        unsigned char pending; /* XON/XOFF will be sent when xmt is free*/
        unsigned char canceled; /* read has been canceled */
        unsigned char flushingRdBuf; /* critical section marker */
    } rdState;
    RING_ID wrtBuf; /* ring buffer for write */
    SEMAPHORE wrtSyncSem; /* writer synchronization semaphore */
    struct /* current state of the write channel */
    {
        unsigned char busy; /* transmitter is busy sending character */
        unsigned char xoff; /* output has been XOFF’d */
        unsigned char cr; /* CR should be inserted next (after LF) */
        unsigned char canceled; /* write has been canceled */
        unsigned char flushingWrtBuf; /* critical section marker */
        unsigned char wrtBufBusy; /* task level writing to buffer */
    } wrtState;
    UINT8 lnNBytes; /* number of bytes in unfinished new line */
    UINT8 lnBytesLeft; /* number of bytes left in incompletely
dequeued line */
    unsigned short options; /* options in effect for this channel */
    FUNCPTTR txStartup; /* pointer to routine to start xmitter */
    FUNCPTTR protoHook; /* protocol specific hook routine */
    int protoArg; /* protocol specific argument */
    SEL_WAKEUP_LIST selWakeupList; /* tasks that are selected on this dev */
} TY_DEV;
```
Each driver uses the TY_CO_DEV data type to define an array, tyCoDv, for the number of channels to be used on the device. Example 6–3 and Example 6–4 show the tyCoDv array structures defined by the Digital AXPvme Z8C530 serial driver, yourCloneTree/config/AXPvme/tyCoDrv.c, and the Digital EB66 PC87312 serial driver, yourCloneTree/config/eb66/tyCoDrv.c, respectively. AXPvme defines 1 channel, EB66 up to 2 channels. When multiple channels are defined, the third and fourth fields provide the channel number (normally numbered upwards from 0) and the interrupt vector.

Example 6–3  Z8C530 Serial Driver Array Structure tyCoDv

```c
LOCAL TY_CO_DEV tyCoDv [NUM_TTY];
```

Example 6–4  PC87312 Serial Driver Array Structure tyCoDv

```c
LOCAL TY_CO_DEV tyCoDv[NUM_TTY] =
{
  { { (NULL)}}, FALSE, PC87312_SERIAL_PORT_1, ISA_INT_SERIAL_PORT_1),
#if (NUM_TTY > 1)
  { { (NULL)}}, FALSE, PC87312_SERIAL_PORT_2, ISA_INT_SERIAL_PORT_2),
#endif
};
```

6.3 Driver Operations

Table 6–3 lists the routines common to most or all implementations of the VxWorks serial driver, **tyCoDrv**.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tyCoDevCreate()</td>
<td>Create a device for an onboard serial port</td>
</tr>
<tr>
<td>tyCoDrv()</td>
<td>Initialize the serial driver</td>
</tr>
<tr>
<td>tyCoInt()</td>
<td>Handle serial device interrupts</td>
</tr>
<tr>
<td>tyCoIoctl()</td>
<td>Handle I/O control requests</td>
</tr>
</tbody>
</table>

(continued on next page)
tyCoDevCreate() and tyCoDrv() are driver initialization routines; they are user callable, but typically are called from system initialization code during VxWorks system startup. Several example implementations of these routines are documented in the VxWorks hardware supplements and the corresponding reference pages, target_tyCoDrv and target_tyCoDevCreate().

tyCoIoctl() and tyCoOpen() are local routines that are identified to, and accessed by, the I/O system.

tyCoStartup() is a local routine that is accessed through a procedure handle supplied to the TY_DEV data structure at driver initialization.

ntyCoInt() is an interrupt handler routine; it is accessed through an interrupt vector that is connected using the intConnect() routine during driver initialization.

Other routines can be created as needed; for example, the Motorola MVME147S-1 BSP defines local routines tyCoResetChannel() and tyCoInitChannel() to reset and initialize a single channel (among two channels available).

\section*{tyCoDevCreate()}

Create a device for an onboard serial port

\textbf{C Syntax}

\begin{verbatim}
STATUS tyCoDevCreate (char *name,
                   int channel,
                   int rdBufSize,
                   int wrtBufSize);
\end{verbatim}
Serial Driver Operations

Arguments

`name`
Specifies the name to use for this device.

`channel`
Specifies the physical channel for this device. Valid values vary between controllers and boards implementing them; typically one value is 0 for a first channel, another 1 (or 4) for a second channel, and so on.

`rdBufSize`
Specifies the read buffer size, in bytes.

`wrtBufSize`
Specifies the write buffer size, in bytes.

Description

This routine creates a device on a specified serial port. Each port you use should have exactly one device associated with it, by calling this routine.

For example, to create the device "/tyCo/0" with buffer sizes of 512 bytes, the proper call would be:

```
tyCoDevCreate ("/tyCo/0", 0, 512, 512);
```

Normally, this routine is called by `usrRoot()` in yourCloneTree/config/all/usrConfig.c.

This routine is global and should be documented in a target-specific manpage, /usr/opt/VXWnnn/man/man2/target_tyCoDevCreate.2. Use an existing target_tyCoDevCreate.2 manpage as a starting point and modify the manpage, as necessary, to match your modifications to tyCoDrv.c.

Return Values

OK if successful.

ERROR if an error occurred. Possible reasons include:

- The driver is not installed.
- The channel is invalid.
- The device already exists.
tyCoDrv()

Initialize the ty driver

C Syntax

```c
STATUS tyCoDrv();
```

Description

This routine initializes the driver, sets up interrupt vectors, and performs hardware initialization of the serial ports.

This routine should be called exactly once, before any reads, writes, or calls to `tyCoDevCreate()`. Normally, it is called by `usrRoot()` in `yourCloneTree/config/all/usrConfig.c`.

You should modify this routine to connect the interrupt routines to the proper interrupt vectors for your target board.

This routine is global and should be documented in a target-specific manpage, `/usr/opt/VXWnnn/man/man1/target_tyCoDrv.1`. Use an existing `target_tyCoDrv.1` manpage as a starting point and modify the manpage, as necessary, to match your modifications to `tyCoDrv.c`.

Return Values

- OK if successful.
- ERROR if the driver cannot be installed.

---

tyCoInt()

Handle serial device interrupts

C Syntax

```c
VOID tyCoInt();
```
Serial Driver Operations

Description
This routine handles serial device interrupts. Typically, it is connected to an
interrupt vector at initialization time and is accessed through the vector when
a device interrupt occurs.

Interrupt handler routines must be defined in a target-specific manner. For
example, the MVME147S-1 BSP defines tyCoInt() to handle interrupt-
level processing for both available channels, dispatching to other routines
as appropriate. Other drivers establish separate interrupt handlers for
transmitter and receiver interrupts.

For a receive interrupt, the interrupt handler code calls the tyIRd() routine
in tyLib to add the character to the input buffer, and resets the interrupt as
appropriate.

For a transmit interrupt, the interrupt handler code calls the tyITx() routine
in tyLib to get a character from the transmit buffer, and resets the interrupt
as appropriate.

---

tyCoIoctl()  
Handle I/O control requests

C Syntax
LOCAL STATUS tyCoIoctl(TY_CO_DEV *pTyCoDv,
int request,
long arg);

Arguments

pTyCoDv
Specifies the device to control.

request
Specifies the ioctl() request code.

arg
Specifies an argument, if any, appropriate to the request type.
Serial Driver Operations

Description

The `tyCoIoctl()` routine handles `ioctl()` requests for the serial device. In most cases, it handles the FIOBAUDRATE request, setting the requested baud rate, and passes all others to the `tyIoctl()`. (For details on other requests handled, see the reference pages for `tyIoctl()` and the `tyLib` library.)

This routine is not called directly by user code.

Return Values

OK if successful.
ERROR if invalid baud rate for FIOBAUDRATE request.
The value returned by `tyIoctl()` for requests other than FIOBAUDRATE.

---

**tyCoOpen()**

Open a file to the serial device

**C Syntax**

```c
LOCAL long tyCoOpen (TY_CO_DEV *pTyCoDv,
                     char *name,
                     int mode);
```

**Arguments**

- `pTyCoDv`
  Specifies the file pointer to be returned to the I/O system caller.

- `name`
  Not used by this routine.

- `mode`
  Not used by this routine.

**Description**

The `tyCoOpen()` routine returns a file pointer for the serial device to the I/O system caller.

This routine is not called directly by user code.
Serial Driver Operations

Return Value
File pointer pTyCoDv.

\textbf{tyCoStartup()}

Start the transmitter

\textbf{C Syntax}

\begin{verbatim}
LOCAL VOID tyCoStartup (TY_CO_DEV *pTyCoDv);
\end{verbatim}

Argument

\texttt{pTyCoDv}

Specifies the serial device to start up.

Description

The routine \texttt{tyCoStartup()} is used to start the transmitter. Typically it calls the \texttt{tyLib} routine \texttt{tyITx()} to get a character from the transmit buffer to transmit. After this routine executes, the transmitter is enabled. Subsequent transmits are performed in response to device interrupts by the interrupt routine.

This routine is not called directly by user code, but is accessed by the I/O system through a pointer in the device's \texttt{TY_DEV} data structure.
6.4 Sample Serial Device Driver

Example 6–5 provides a listing of the driver for the Z8C530 serial controller, a controller chip used on Motorola MVME147S-1 boards as well as Digital AXPvme boards. This is the AXPvme implementation.

You can also access, on line, the serial driver source modules in /config/target/tyCoDrv.c and /src/drv/serial.

Example 6–5  Z8C530 Serial Driver Listing

/* tyCoDrv.c - DEC Medulla Z8c530 serial line handler */
/*
  Copyright (c) 1992
  DIGITAL Equipment Corporation, Maynard, Mass.
*/
/* Copyright 1984-1990 Wind River Systems, Inc. */
extern char copyright_wind_river[]; static char *copyright=copyright_wind_river;

/*
DESCRIPTION
This is the driver for the DEC Medulla on-board serial port.
It uses the SCC’s only in asynchronous mode.

USER CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O
system. Two routines, however, must be called directly, tyCoDrv to
initialize the driver, and tyCoDevCreate to create devices.

TYCODRV
Before using the driver, it must be initialized by calling the routine:
  .CS
  tyCoDrv ()
  .CE
This routine should be called exactly once, before any reads, writes, or
tyCoDevCreates. Normally, it is called from usrRoot.

CREATING TERMINAL DEVICES
Before a terminal can be used, it must be created. This is done
with the tyCoDevCreate call.
  .CS
  STATUS tyCoDevCreate (name, channel, rdBufSize, wrtBufSize)
    char *name;     /* Name to use for this device */
    int channel;    /* Physical channel for this device (0..1) */
    int rdBufSize;  /* Read buffer size, in bytes */
    int wrtBufSize; /* Write buffer size, in bytes */
  .CE

(continued on next page)
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

For instance, to create the device "/tyCo/0", with buffer sizes of 512 bytes, the proper call would be:

```c
    tyCoDevCreate("/tyCo/0", 0, 512, 512);
```

IOCTL
This driver responds to all the same ioctl codes as a normal ty driver. Baud rates between 30 and 9600 baud is available. */

```
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "tyLib.h"
#include "drv/serial/z85c30.h"
#include "config.h"
#include "arch/alpha/kdebug.h"
#include "adpLib.h"

#define SETTLE_SCC (int i; for(i=0; i<7; i++) mb();)
#define SCC_WRITE_RAP( reg, val) { *cr = ((volatile u_int)(reg)); 
    mb(); 
    *cr= ((volatile u_int)(val)); 
    mb(); 
}
#define SCC_READ_RAP(reg,val) { *cr=(volatile u_int)(reg); 
    mb(); 
    val=((volatile u_int)(*cr))&0xff; 
}
#define SCC_RESET_RAP( val) { (val) = *cr; 
    mb(); 
    (val) = *cr; 
}
#define SCC_READ_DATA( val) { mb(); 
    val = ((volatile u_int)(*dr) ) &0xff; 
}
#define SCC_WRITE_DATA(val){ mb(); 
    *dr = ((volatile u_int)(val)); 
    mb(); 
}

LOCAL TY_CO_DEV tyCoDv [NUM_TTY];
LOCAL int tyCoDrvNum=0; /* driver number assigned to this driver */
```

(continued on next page)
Example 6–5 (Cont.)  Z8C530 Serial Driver Listing

/* forward declarations */
LOCAL VOID tyCoStartup ();
LOCAL long tyCoOpen ();
LOCAL STATUS tyCoIoctl ();
VOID tyCoInt ();
VOID tyCoHrdInit ();
LOCAL VOID tyCoInitChannel ();
LOCAL VOID tyCoResetChannel ();
extern KdebugInfo *kdebug_info;
extern struct adp *adpPCI;
extern int cpuMhz;
extern struct sysIoBlock sysIoBlock;

/******************************************************************************
* tyCoDrv - ty driver initialization routine
* This routine initializes the serial driver, sets up interrupt vectors,
* and performs hardware initialization of the serial ports.
* RETURNS: OK or ERROR if unable to install driver.
*/
STATUS tyCoDrv ()
{
    /* check if driver already installed */
    if (tyCoDrvNum > 0)
        return (OK);
    tyCoDrv[0].cr = (u_int *) PCI_IO (SCCA_RAP, ABYTE);
    tyCoDrv[0].dr = (u_int *) PCI_IO (SCCA_DATA, ABYTE);
#if (NUM_TTY > 1)
    tyCoDrv[1].cr = (u_int *) PCI_IO (SCCB_RAP, ABYTE);
    tyCoDrv[1].dr = (u_int *) PCI_IO (SCCB_DATA, ABYTE);
#endif
    if (adpIntConnect (PCI_VECTOR(0,4), 0, tyCoInt, NULL, NULL ) == ERROR)
        return ERROR;
    tyCoDrvNum = iosDrvInstall ((FUNCPTR)tyCoOpen, (FUNCPTR) NULL,
                              (FUNCPTR)tyCoOpen, (FUNCPTR) NULL,
                              (FUNCPTR)tyRead, (FUNCPTR)tyWrite,
                              (FUNCPTR)tyCoIoctl);
    tyCoHrdInit(0); /* ycl setup sw before start hw */
}

(continued on next page)
# if (NUM_TTY > 1) /* init second port for terminal */
    tyCoHrdInit(1);
#endif

return (tyCoDrvNum == ERROR ? ERROR : OK);
}

/*******************************************************************************
* tyCoDevCreate - create a device for the onboard ports
* This routine creates a device on one of the serial ports. Each port
* to be used should have exactly one device associated with it, by calling
* this routine.
* RETURNS: OK or ERROR if driver not installed or invalid channel.
*/

STATUS tyCoDevCreate (name, channel, rdBufSize, wrtBufSize)
char *name; /* Name to use for this device */
FAST int channel; /* Physical channel for this device (0 or 1) */
int rdBufSize; /* Read buffer size, in bytes */
int wrtBufSize; /* Write buffer size, in bytes */
{
FAST TY_CO_DEV *pTyCoDv;
if (tyCoDrvNum <= 0)
{
    errnoSet (S_ioLib_NO_DRIVER);
    return (ERROR);
}
/* if this doesn’t represent a valid channel, don’t do it */
if (channel < 0 || channel >= NUM_TTY)
    return (ERROR);
/* if trying to init channel b, but kdebug is enabled - return ERROR */
if (channel == 1 && kdebug_info->state == KDEBUG_ENABLED)
    return (ERROR);
pTyCoDv = &tyCoDv [channel];
/* if there is a device already on this channel, don’t do it */
if (pTyCoDv->created)
    return (ERROR);
/* initialize the ty descriptor */

(continued on next page)
Example 6–5 (Cont.)  Z8C530 Serial Driver Listing

```c
if (tyDevInit (&pTyCoDv->tyDev, rdBufSize, wrtBufSize, (FUNCPTR)tyCoStartup) != OK)
    return (ERROR);
/* initialize the channel hardware */
tyCoInitChannel (channel);
/* mark the device as created, and add the device to the I/O system */
pTyCoDv->created = TRUE;
if (adpSetBusIntEna (adpPCI, PCI_VECTOR(0,4),0) != OK) /* Enable ints */
    return ERROR;
return (iosDevAdd (&pTyCoDv->tyDev.devHdr, name, tyCoDrvNum));
}
/******************************************************************************
* tyCoHrdInit - initialize the USART
*******************************************************************************/
VOID tyCoHrdInit (int channel)
{
    FAST int oldlevel; /* current interrupt level mask */
    oldlevel = intLock (); /* disable interrupts during init */
tyCoResetChannel (channel); /* reset channel */
    intUnlock (oldlevel);
}
/******************************************************************************
* tyCoResetChannel - reset a single channel
*******************************************************************************/
LOCAL VOID tyCoResetChannel (int channel)
{
    volatile u_int *cr = tyCoDv[channel].cr;
    volatile u_int *dr = tyCoDv[channel].dr;
    volatile u_int tmp;
    SCC_RESET_RAP(tmp);
    sysDelay (10000); /* polled wait, 10 mSec */
    /* if channel = 0, then it's channel a which is the console = 0x80 resets ch. a,
    0x40 resets ch b */
    SCC_WRITE_RAP(WR0_REG9, (channel ? 0x40 : 0x80));
}(continued on next page)
```
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

```c
sysDelay (10000); /* polled wait, 10 mSec */
}
******************************************************************************
* tyCoInitChannel - initialize a single channel
*/
LOCAL VOID tyCoInitChannel (int channel)
{
    volatile unsigned int b_dummy;
    volatile u_int *cr = tyCoDv[channel].cr;
    volatile u_int *dr = tyCoDv[channel].dr;
    volatile int oldlevel;

    oldlevel = intLock();
    /* Do a couple reads to assure we'll be writing WR0. */
    SCC_RESET_RAP(b_dummy);
    /* Be doubly sure these bits get turned off (even though the
     * hardware & software resets are supposed to do it. */
    SCC_RESET_RAP(b_dummy);
    SCC_WRITE_RAP(WR0_REG15, 0x00);
    SCC_RESET_RAP(b_dummy);
    SCC_WRITE_RAP(WR0_REG1, 0x00);
    /* Vendor recommends hitting this one (Ext_Ints) twice! */
    SCC_RESET_RAP(b_dummy);
    SCC_WRITE_RAP(WR0_REG0, WR0_RESET_EXT_INTS);
    SCC_RESET_RAP(b_dummy);
    SCC_WRITE_RAP(WR0_REG2, 0x00);
    /* Set up channel -- */
    /* x16 clock; 1 stop bits; no parity. */
    SCC_WRITE_RAP(WR0_REG4, WR4_CLKMODE_16 | WR4_STOPBITS_1 | WR4_PARITY_NONE);
    /* Rx 8 bits; Rx disabled. */
    SCC_WRITE_RAP(WR0_REG3, WR3_8BITS_RX);
    /* Tx 8 bits; Tx disabled; DTR & RTS on. */
    SCC_WRITE_RAP(WR0_REG5, WR5_8BITS_TX | WR5_DTR | WR5_RTS);
    /* Interrupts disabled. yc add en_msk */
    SCC_WRITE_RAP(WR0_REG9, WR9_MSTR_INT_EN);
    (continued on next page)
```
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

/* in diag code the next is done separately from another routine */
SCC_WRITE_RAP(WR0_REG1, WR1_INT_ALL_RX | WR1_TX_INT_ENABLE);
/* NRZ. */
SCC_WRITE_RAP(WR0_REG10, 0x0);
SCC_WRITE_RAP(WR0_REG11, WR11_RXCLK_BRG_OUT | WR11_TXCLK_BRG_OUT);
if (cpuMhz != 150) {
    SCC_WRITE_RAP(WR0_REG12, 0x32 ); /* 9600 baud, low byte */
    SCC_WRITE_RAP(WR0_REG13, 0x00 ); /* 9600 baud, hi byte */
} else {
    SCC_WRITE_RAP(WR0_REG12, 0x26 ); /* 9600 baud, low byte */
    SCC_WRITE_RAP(WR0_REG13, 0x00 ); /* 9600 baud, hi byte */
}
/* BRG in = RT x C; BRG off. */
SCC_WRITE_RAP(WR0_REG14, WR14_SCC_BRG_SRC);
/* Finally, turn on the SCC channel. */
/* BRG (Baud-Rate Generator) enable*/
SCC_WRITE_RAP(WR0_REG14, WR14_BRG_ENABLE | WR14_SCC_BRG_SRC);
/* Rx Enable. */
SCC_WRITE_RAP(WR0_REG3, WR3_8BITS_RX | WR3_RX_ENABLE);
/* Tx Enable. */
SCC_WRITE_RAP(WR0_REG5, WR5_8BITS_TX | WR5_TX_ENABLE | WR5_DTR | WR5_RTS);
intUnlock(oldlevel); /* Unlock ints */
}

/*******************************************************************************/
/*
* tyCoOpen - open file to USART
*
* ARGUSED0
*/
LOCAL long tyCoOpen (pTyCoDv, name, mode)
    TY_CO_DEV *pTyCoDv;
    char *name;
    int mode;
{
    return ((unsigned long) pTyCoDv);
}

(continued on next page)
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

/*******************************************************************************
* tyCoIoctl - special device control
*******************************************************************************
* This routine handles FIOBAUDRATE requests and passes all others to tyIoctl1.
* RETURNS: OK or ERROR if invalid baud rate, or whatever tyIoctl1 returns.
* /
LOCAL STATUS tyCoIoctl (pTyCoDv, request, arg)
    TY_CO_DEV *pTyCoDv; /* device to control */
    int request; /* request code */
    long arg; /* some argument */
{
    FAST int oldlevel; /* current interrupt level mask */
    FAST int baudConstant;
    FAST STATUS status;
    volatile u_int *cr; /* SCC control reg adr */
    switch (request)
    {
    case FIOBAUDRATE:
        if (arg < 50 || arg > 204200)
        {
            status = ERROR; /* baud rate out of range */
            break;
        }
        /* Calculate the baud rate constant for the new baud rate
         * from the input clock frequency.  This assumes that the
         * divide-by-16 bit is set in the Z8530 WR4 register (done
         * in tyCoInitChannel).
         */
        if (cpuMhz == 150)
            baudConstant = (12500000 / (2 * arg * 16)) - 2;
        else
            baudConstant = (16000000 / (2 * arg * 16)) - 2;
        cr = pTyCoDv->cr;
        oldlevel = intLock ();
        SCC_WRITE_RAP(WR0_REG12, baudConstant & 0xff);
        SCC_WRITE_RAP(WR0_REG13, baudConstant >> 8 & 0xff);
        intUnlock (oldlevel);
        status = OK;
        break;
    (continued on next page)
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

```c
default:
    status = tyIoctl (&pTyCoDv->tyDev, request, arg);
    break;
}
return (status);
}

/*******************************************************************************
* tyCoInt - interrupt level processing
* This routine handles interrupts from the SCC. It is called from the main
* system int handler.
*/
VOID tyCoInt (channel)
    int channel;
{
    volatile u_int temp, intStatus;
    TY_CO_DEV *pTyCoDv;
    char Char;
    volatile u_int *dr;
    volatile u_int *cr;
    cr = (volatile u_int *) PCI_IO(SCCA_RAP,0); /* Must read channel A */
    SCC_READ_RAP(WR0_REG3, intStatus); /* get the interrupt status from ch. a
    (ch. a has status of both channels */
    if (intStatus & 0x38){ /* any bit for ch. a */
        pTyCoDv = &tyCoDv [0]; /* Point to channel A */
        dr = pTyCoDv->dr; /* point to ch. a data reg. */
        if (intStatus & RR3_CHANNELA_TXIP){ /* do we have a channel a tx int. ? */
            if (pTyCoDv->created && (tyITx (&pTyCoDv->tyDev, &Char) == OK)){
                SCC_WRITE_DATA(Char);
            }
            else{
                /* No more chars to xmit now. reset the tx int,
                so the SCC doesn’t keep interrupting us. */
                SCC_WRITE_RAP(WR0_REG0, WR0_RST_TX_INT);
            }
        }
        if (intStatus & RR3_CHANNELA_RXIP) { /* do we have a channel a rx int. ? */
            /* (continued on next page)
        }
```
if (pTyCoDv->created){
    SCC_READ_DATA(temp); /* read char into temp */
    Char = (char) (temp & 0x00ff); /* use Char to hold the */
    tyIRd (pTyCoDv->tyDev, Char); /* received char */
}

if (intStatus & RR3_CHANNELA_EXTSTAT){ /* channel a ext/stat int. ? */
    SCC_WRITE_RAP(WR0_REG0, WR0_ERR_RST);
    SCC_READ_DATA(temp); /* throw away the char */
}
/* if a ch a. interrupt */

if (intStatus & 0x7){ /* any bit for ch. b? */
    if (NUM_TTY < 1 || kdebug_info->state == KDEBUG_ENABLED) {
        sysDelay (10000);
        prom_puts ("tyCoInt: cannot service channel B interrupt\n\n");
        halt();
    }
    pTyCoDv = &tyCoDv[1]; /* Point to channel B */

    cr = (volatile u_int *) PCI_IO(SCCB_RAP,0); /* Must read channel A */
    dr = pTyCoDv->dr; /* point to ch. B data reg. */
    if (intStatus & RR3_CHANNELB_TXIP){ /* do we have a channel b tx int. ? */
        (continued on next page)
Example 6–5 (Cont.) Z8C530 Serial Driver Listing

```c
if (pTyCoDv->created && (tyITx (&pTyCoDv->tyDev, &Char) == OK)){
    SCC_WRITE_DATA(Char);
} else{
    /*
     * No more chars to xmit now. reset the tx int,
     * so the SCC doesn't keep interrupting us.
     */
    SCC_WRITE_RAP(WR0_REG0, WR0_RST_TX_INT);
}

if (intStatus & RR3_CHANNELB_RXIP){ /* do we have a channel b rx int. ? */
    if (pTyCoDv->created){
        SCC_READ_DATA(temp); /* read char into temp */
        Char = (char) (temp & 0x00ff); /* use Char to hold the */
        tyIRd (&pTyCoDv->tyDev, Char); /* received char */
    }
}

if (intStatus & RR3_CHANNELB_EXTSTAT){ /* do we have a channel a ext/stat int. ? */
    SCC_WRITE_RAP(WR0_REG0, WR0_ERR_RST);
    SCC_READ_DATA(temp); /* throw away the char */
}
} /* if a ch b interrupt */
/* end int. routine */
ологической строки*/

*******************************************************************************
* tyCoStartup - transmitter startup routine
* Call interrupt level character output routine.
*/
LOCAL VOID tyCoStartup (pTyCoDv)
TY_CO_DEV *pTyCoDv; /* ty device to start up */
{
    char outChar;
    volatile u_int *dr;
    if (tyITx (&pTyCoDv->tyDev, &outChar) == OK){
        dr = pTyCoDv->dr;
        SCC_WRITE_DATA(outChar);
    }
}
This chapter explains how to write a device driver for a network interface device that is not already supported by VxWorks for Alpha. It describes the routine interfaces that connect a driver to higher-level network communication protocols, including the following:

- NFS client
- \texttt{ftp} client and server
- \texttt{rlogin} client and server
- X-windows client
- \texttt{telnet} client
- Remote source debugger (VxGDB)
- Remote command execution
- RPC
- BSD sockets

The architecture of VxWorks network components is shown in Figure 7–1. Network device drivers implement the network interface layer, which connects the network communication protocols to the physical network.

Typically, VxWorks uses Ethernet as its basic network transmission medium, but the Internet protocols can also run over serial lines or shared memory on a common backplane. For more information on high-level network facilities or low-level transmission media, see the Network chapter of the VxWorks Programmer’s Guide.
If you are porting VxWorks to a board containing a network chip, first determine if VxWorks already supports a driver for the network chip on the board. If your chip is supported, you do not need to write a new driver; rather, you modify board-specific `sys*` routines that handle board-dependent aspects of the network chip’s operation. For example, if your board contains a LANCE chip, supported by the `if_ln` driver, you modify `sysLnAllocateMem()` and other BSP routines called by the driver to perform appropriate board-specific actions. If your board contains a DECchip 21040 Ethernet interface chip, supported by the `if_tu` driver, you check that `sysEnetAddrGet()` and other BSP routines called by the driver perform appropriate board-specific actions. In most cases, the existing network driver should require very limited, or no, modification for your target configuration.

If VxWorks does not support a driver for your network chip, you write a driver using this chapter, a network driver template, and BSP source files.
Writing a network driver includes the following steps:

1. Establish a network driver template, using the driver listing in Section 7.5 and the driver template located at yourCloneTree/src/demo/AXPenp. If you purchased a VxWorks Board Porting Kit or Source Kit, you can additionally refer to the network driver source modules in yourCloneTree/src/drv/netif.

2. Select a driver mnemonic for your device, to be used in the driver name, the names of required driver routines, and driver definitions. For example, the DECchip 21040 Ethernet driver uses tu (if_tu.c, tuattach(), tuOutput(), #ifdef INCLUDE_TULIP).

3. Code the well-defined set of device-specific driver routines expected by higher-level network protocols, described in Section 7.3.

4. As part of coding required driver routines, you will need to define device-specific data structures, described in Section 7.2, in a header file named after your driver. (For example, if_tu.h.)

5. Place device address and vector definitions needed by initialization and driver routines in your BSP’s config.h file, or in the target.h file included by config.h. Also, check that support for your network device is enabled in config.h.

6. If you want your driver to be readily portable across multiple platforms using the same network controller chip, you should code BSP hook routines to handle board dependencies. For example, the LANCE Ethernet driver implements callouts to sysLnAllocMem() and several other routines, ensuring the driver is portable across Motorola MVME147S-1, EB64, and other (currently-unsupported) platforms. The DECchip 21040 Ethernet driver calls BSP routines such as sysEnetAddrGet(), sysDelay(), and sysClkRateGet() to request board-dependent information or actions. In addition to these routines, you should consider whether adding callouts from the driver to your BSP would promote the board-independence of your driver code.

7. Edit the system-dependent network initialization routine, usrNetInit(), in module yourCloneTree/src/config/usrNetwork.c, to ensure that your network driver is included in the system’s network initialization path.

8. If you want to be able to boot from your network device, you must build your driver into the VxWorks boot ROM. Edit the module yourCloneTree/config/all/bootConfig.c to include your driver. Follow what is done for other drivers, such as adding an entry in the netif structure and IMPORTing the driver attach routine.
This chapter addresses the following topics:

- Driver overview, Section 7.1
- Driver data structures, Section 7.2
- Driver operations, Section 7.3
- Network support routines, Section 7.4
- Sample network interface driver, Section 7.5

### 7.1 Driver Overview

Network device drivers constitute the lowest level of VxWorks network software. Network I/O requests originate from high-level network facilities such as `rlogin`, `telnet`, VxGDB, NFS, RPC, `rsh`, and `ftp`, and are communicated through BSD sockets and Internet protocols (TCP/IP and UDP/IP) to the network device driver.

The organization of the network interface driver does not fit the model for a standard VxWorks I/O driver. Network drivers are not accessible through the regular I/O driver interface, involving the use of `open()` and other file routines. Network drivers are designed to meet the requirements of the higher-level communication protocols.

A network interface device driver is expected to provide a well-defined set of device-specific routines, described in Section 7.3, for higher-level network protocols to access. These include `attach()`, `Init()`, `Ioctl()`, `Output()`, `Reset()`, and `Int()` routines, each name prefixed with a device mnemonic.

For example, the DEChip 21040 Ethernet driver, `if_tu`, provides the following routines for upper-level network protocols:

```c
    tuattach()
    tuIoctl()
    tuOutput()
    tuReset()
    tuInt()
```

Note

The `if_tu` driver performs all initialization within the `tuattach()` routine; thus, no separate `tuInit()` routine is provided.
Table 7–1 shows the initialization and runtime flow of control of a network interface driver.

**Table 7–1  Network Interface Driver Flow of Control**

<table>
<thead>
<tr>
<th>Initialization</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>usrRoot()</td>
<td>⇒</td>
<td>usrNetInit()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gets device name and Internet address, and adds the device to the system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⇒ xxattach()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Network⇒</td>
<td>xxIoctl(), other driver routines</td>
<td>if_xx.c (driver)</td>
</tr>
<tr>
<td></td>
<td>Handle device dependencies, call BSP as necessary for board dependencies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>⇒ sys* routines</td>
<td>sysLib.c</td>
</tr>
<tr>
<td>Device⇒</td>
<td>xxInt() ISR</td>
<td>if_xx.c (driver)</td>
</tr>
<tr>
<td></td>
<td>Handles device interrupts</td>
<td></td>
</tr>
</tbody>
</table>

### 7.1.1 Attaching the Driver to the System

The information needed for upper-level network protocols to access an network device driver is provided in the driver's ifnet data structure, defined in yourCloneTree/net/if.h and described in Section 7.2. The network driver's attach() routine, called at system initialization in the *usrConfig* or *bootConfig* module, fills in fields of the ifnet structure with the names of the appropriate Init(), Ioctl(), Output(), and Reset() routines, and enters the driver's ifnet structure in the system's list of available network devices. The driver's routines can then be accessed as needed by upper-level network protocols, using generic procedure handles defined by the ifnet structure.

### 7.1.2 Interrupt Handling

The driver's interrupt handler routine, xxInt(), is accessed, in response to a device interrupt, through an interrupt vector. At system startup, the driver's xxattach() routine connects the interrupt handler to a vector using the intConnect() or adpIntConnect() routine, described in VxWorks user documentation. The connect routines each accept a parameter to be passed to the interrupt handler. When a device interrupt occurs, xxInt() is called with the parameter you specified.
The interrupt handler takes the appropriate action to handle the interrupt. Typically this involves performing as little processing as possible at interrupt level, and deferring more time-consuming processing for execution at task level by the network task, `netTask()`. See the formal description of `xxInt()` for more details.

### 7.1.3 Support Routines

To implement the driver-specific functions expected by upper-level network protocols, the driver can make use of network support routines and macros provided by network libraries or header files. For example, the driver’s `attach()` routine can use the `ether_attach()` routine to do much of the work of filling in the `ifnet` structure and attaching the network device to the system. Also, the driver’s interrupt handler can use the `netJobAdd()` routine to queue an interrupt-processing function to the job queue of the network task, `netTask()`, for later processing at task level. Network support routines are described in Section 7.4.

### 7.1.4 Ethernet Hook Routines

VxWorks provides `etherLib`, a library of raw I/O routines and hooks, to allow applications and drivers to access raw Ethernet packets, bypassing protocol layers such as TCP. This library and its routines are described by VxWorks user documentation, including reference pages for `etherLib`, `etherInputHookAdd()`, and `etherOutputHookAdd()`.

A call to `etherInputHookRtn()` should be invoked in the receive routine of every network driver providing this service, before passing a new input packet to a higher network layer. For example:

```c
#include <etherLib.h>

void xxRecv ()
{
    /* call input hook if any */
    if ((etherInputHookRtn != NULL) &&
        (* etherInputHookRtn) (&ls->ls_if, (char *)eh, len))
    {
        return; /* input hook has already processed this packet */
    }
```

If the hook routine returns a non-zero value, the input frame was processed by the hook routine and should not be passed up to the network layer.
Similarly, a call to etherOutputHookRtn() should be invoked in the transmit routine of every network driver providing this service, before sending an output packet over the Ethernet. For example:

```c
#include <etherLib.h>
...
xxStartOutput ()
/* call output hook if any */
if ((etherOutputHookRtn != NULL) &&
    (* etherOutputHookRtn) (&ls->ls_if, buf0, len))
    { /* output hook has already processed this packet */
    } else
    ...
```

If the hook routine returns a non-zero value, the output frame was processed by the hook routine and should not be sent over the Ethernet.

7.1.5 System-Dependent Network Support Routines

To allow for target-specific configurations of a particular controller chip, and to promote portability of your network driver across multiple target platforms using your network controller chip, some routines may need to be modified or added to the system-dependent routines library, sysLib.

7.2 Driver Data Structures

Each network device driver defines a drv_ctrl structure, which contains driver-specific information. The drv_ctrl structure includes other important driver data structures, including the arpcmb (defined locally as IDR) and ifnet structures. The structure is allocated and initialized during system startup by the driver attach routine (see Section 7.3).

Example 7-1 shows the drv_ctrl structure defined for the DECchip 21040 Ethernet driver in yourCloneTree/h/drv/netif/if_tu.h. Some related value and type definitions are omitted from this example, but are available in the header file.
Example 7–1  DECchip 21040 Ethernet Driver Data Structure *drv_ctrl*

typedef struct arpcom  IDR; /* Interface Data Record wrapper */
typedef struct ifnet IFNET; /* real Interface Data Record */

typedef struct drv_ctrl {
  IDR idr; /* Interface Data Record */
  struct {
    TUDESC *ring; /* receive ring base addr */
    TU_DATABUF *bufs; /* receive buffer base */
    int index; /* index into rx ring & buffer */
  } rx_ring;
  struct {
    TUDESC *ring; /* transmit ring base addr */
    TU_DATABUF *bufs; /* transmit buffer base */
    int index; /* index into tx ring & buffer */
  } tx_ring;
  int reclaimIndex; /* reclaim index into tx ring */
  int numRxDesc; /* # of rx descriptors in ring */
  int numTxDesc; /* # of tx descriptors in ring */
  SEM_ID TxSem; /* transmitter semaphore */
  SEM_ID RxSem; /* rcvr semaphore */
  BOOL attached; /* indicates unit is attached */
  u_char flags; /* Misc control flags */
  int adpVec; /* interrupt vector (adp vector) */
  u_long pciMemStart; /* start of our memory pool, PCI addr */
  char *memStart; /* start of our memory pool, KSEG */
  char *memEnd; /* last used addr of our mem pool */
  ADAPTER *pAdp; /* ptr to tulip's adapter */
  void *obMapObj; /* outbound path map obj handle */
  u_long obSrcAdr; /* srcAdr of outbound path (KSEG addr) */
  u_long obMapSize; /* size of outbound mapping */
  u_long obMapGran; /* granularity of outbound mapping */
  void *ibMapObj; /* inbound path map obj handle */
  u_long ibSrcAdr; /* srcAdr of inbound path (PCI addr) */
  u_long ibMapSize; /* size of inbound mapping */
  char *ibMapStart; /* KSEG addr of start of inbound mapping */
  u_long adpRWflags; /* flags to pass to adpRead/write */
  u_int devAdrs; /* PCI address of csr's */
} (continued on next page)
Example 7–1 (Cont.) DEChip 21040 Ethernet Driver Data Structure `drv_ctrl`

```c
u_long totInts; /* count of tu interrupts */
u_long restartCount; /* count of tu restarts */
u_long csr5Errs; /* count of csr5 errors */
u_long missedRxFrames; /* count of missed rx frames */
TU_MULTI multicast[TU_NMULTI]; /* multicast addresses */
u_long rxOflow; /* receive status counters */
u_long rxCRCErr;
u_long rxLateColl;
u_long rxTooLong;
u_long rxRunt;
u_long rxWontFit;
u_long rxUnavail;
u_long rxNoMbufs; /* can’t bcopy to mbuf */
u_long txUflow; /* transmit status counters */
u_long txUnavail;
u_long txLateColl;
u_long txExcessColl;
u_long txWait;
} DRV_CTRL;
```

The first field in each `drv_ctrl` structure is the `arpcom` (or IDR) structure, shared between network device drivers and network address resolution code. Example 7–2 shows the definition of `arpcom`, from `yourCloneTree/h/netinet/if_ether.h`.

Example 7–2 Network Driver Data Structure `arpcom`

```c
struct arpcom {
    struct ifnet ac_if; /* network-visible interface */
    u_char ac_enaddr[6]; /* ethernet hardware address */
    struct in_addr ac_ipaddr; /* copy of ip address- XXX */
};
```

The first field in the `arpcom` structure is the driver’s `ifnet` structure, which provides the interface between the network device driver and the upper-level network protocols. The information in the `ifnet` structure includes the procedure handles with which the network modules access the required set of driver routines. Example 7–3 shows the definition of `ifnet`, from `yourCloneTree/h/net/if.h`.
Example 7–3 Network Driver Data Structure *ifnet*

\begin{verbatim}
struct ifnet {
    char *if_name; /* name, e.g. `en'' or `lo'' */
    short if_unit; /* sub-unit for lower level driver */
    short if_mtu; /* maximum transmission unit */
    short if_flags; /* up/down, broadcast, etc. */
    short if_timer; /* time 'til if_watchdog called */
    struct ifaddr *if_addrlist; /* linked list of addresses per if */
    struct ifqueue {
        struct mbuf *ifq_head;
        struct mbuf *ifq_tail;
        int ifq_len;
        int ifq_maxlen;
        int ifq_drops;
    } if_snd; /* output queue */
    /* procedure handles */
    int (*if_init)(); /* init routine */
    int (*if_output)(); /* output routine */
    int (*if_ioctl)(); /* ioctl routine */
    int (*if_reset)(); /* bus reset routine */
    int (*if_resolve)(); /* bus reset routine */
    void (*if_watchdog)(); /* timer routine */
    /* generic interface statistics */
    int if_ipackets; /* packets received on interface */
    int if_ierrors; /* input errors on interface */
    int if_opackets; /* packets sent on interface */
    int if_oerrors; /* output errors on interface */
    int if_collisions; /* collisions on csma interfaces */
    /* end statistics */
    struct ifnet *if_next;
};
\end{verbatim}

7.3 Driver Operations

Table 7–2 lists the routines common to most or all implementations of a VxWorks network driver.
Table 7–2 Network Driver Required Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxattach()</td>
<td>Attach network device xx to the network</td>
</tr>
<tr>
<td>xxInit()</td>
<td>Initialize network device xx</td>
</tr>
<tr>
<td>xxInt()</td>
<td>Handle an interrupt for device xx</td>
</tr>
<tr>
<td>xxIoctl()</td>
<td>Handle ioctl() control commands for network device xx</td>
</tr>
<tr>
<td>xxOutput()</td>
<td>Transmit a frame over the network for network device xx</td>
</tr>
<tr>
<td>xxReset()</td>
<td>Reset network device xx</td>
</tr>
<tr>
<td>xxWatchdog()</td>
<td>Perform timer functions for the network device xx driver</td>
</tr>
</tbody>
</table>

\[ xxattach() \] is a driver initialization routine; it is user callable, but typically is called from \texttt{usrConfig} or \texttt{bootConfig} during VxWorks system startup. The driver attach routine fills in the driver’s ifnet structure, described in Section 7.2, and makes the driver accessible to higher-level network protocols. Several example implementations are documented in the VxWorks reference material, including:

- eglattach() Interphase Eagle 4207 interface
- eiattach() Intel 83596 interface
- lnattach() LANCE interface
- tuattach() DECchip 21040 Ethernet interface

\[ xxInit(), xxIoctl(), xxOutput(), xxReset(), and xxWatchdog() \] are local routines that are identified to, and accessed by, higher-level network protocols. Their names are filled into the ifnet data structure by \[ xxattach() \].

\[ xxInt() \] is an interrupt handler routine; it is accessed through an interrupt vector that is connected using the \texttt{intConnect()} routine during driver initialization.

**Note**

It is fairly typical for network drivers to provide a \[ xxShow() \] routine, which is callable by the user (global) but often undocumented, for displaying information about the current state of the network interface at runtime.
Network Driver Operations

The following sections describe each routine and the typical actions it performs. These descriptions are generalized and inclusive, based on network device drivers currently provided by VxWorks; not all the actions described apply to all network hardware.

In this section, the notation “xx” or “XX” is used to represent the network device's two (or three) character mnemonic in routine and structure names.

---

**xxattach()**

Add network device xx to the network

**C Syntax**

```c
STATUS xxattach (int unit,
    char *devAdrs,
    int ivec,
    int ilevel,
    char *memAdrs,
    int spare1,
    int spare2,
    int spare3,
    int spare4);
```

**Arguments**

- **unit**
  Specifies the network device logical unit number (integer value).

- **devAdrs**
  Specifies the device I/O address.

- **ivec**
  Specifies the interrupt vector.

- **ilevel**
  Specifies an interrupt level. (Since this argument is not applicable to current DECchip 21040 implementations, the if_tu driver uses a memSize argument, specifying the number of bytes of interface memory pool to allocate, in this position.)

- **memAdrs**
  Specifies the address of the interface memory pool; NONE (-1) means to allocate the memory pool with malloc().

---

7–12 Writing a Network Interface Driver
Network Driver Operations

Optionally specify other driver-specific parameters, such as data port width, data path mapping flags, and so on.

Description

This routine attaches the network device xx to the network, if the device exists. The routine makes the device available by filling in the network interface (ifnet) record and adding it to the system list of available network devices. The driver is flagged as inactive, and must be initialized using the driver’s Init() routine before the driver is used for I/O. The system will initialize the device when it is ready to accept packets.

The routine may do additional work, such as allocating memory for purposes you define.

The driver attach routine normally is called at system initialization, from the
usrConfig or bootConfig module, particularly if the device is required for booting.

This routine is global and should be documented in a target-independent manpage, /man/man2/xxattach.2. Use an existing xxattach.2 manpage as a starting point and modify the manpage as necessary to match the version in your if_xx driver. Also, use an existing if_xx.1 manpage to create a library overview manpage, describing the general functionality of your network driver.

Return Values

OK if successful.
ERROR if not successful.

See Also

Network Driver Operations

**xxInit()**

Initialize network device xx

**C Syntax**

```
#include <if.h>
#include <if_ether.h>

LOCAL STATUS xxInit (int unit);
```

**Argument**

- **unit**
  Specifies the network device unit number (integer value).

**Description**

This routine, referenced by the procedure handle if_init, initializes the Ethernet device. Typically, it marks the device as inactive; then clears any pending operations, resets, configures, and initializes the device; then marks the device as active.

This routine is not called directly by user code.

**Note**

The if_tu and if_in drivers perform all initialization within their respective xxattach() routines; thus, no separate tuInit() or lnInit() routines are provided and NULL function pointers are specified for the if_init procedure handle.

**Return Values**

- **OK** if successful.
- **ERROR** if not successful.
Network Driver Operations

**xxInt()**

Handle an interrupt for device xx

**C Syntax**

```c
void xxInt (DRV_CTRL *pDrvCtrl);
```

**Argument**

*unit*

Specifies the drv_ctrl data structure (described in Section 7.2) for the interface.

**Description**

This routine, which is invoked upon the occurrence of a device interrupt, handles the interrupt in an appropriate fashion.

Because the interrupt handler executes at interrupt level, with other interrupts locked out, processing within the handler should be kept to a minimum. Any more time-consuming interrupt processing should be queued for execution at task level. You use the netJobAdd() routine to queue driver-defined functions and arguments to the job queue for the network task, netTask(), which dequeues functions and executes them in a task context. The netJobAdd() routine is described in Section 7.4, and the network task is described in the VxWorks user documentation, including netLib and netTask() reference pages.

If the device is on the VMEbus and does not implement automatic acknowledgement of VMEbus interrupts, this routine must manually acknowledge the VMEbus interrupt with a call to the sysBusIntAck() routine.

This routine is accessed through an interrupt vector and should not be called by user code.
Network Driver Operations

**xxioctl()**

HANDLE ioctl() control commands for network device xx

**C Syntax**

```c
#include <if.h>
#include <if_ether.h>
int xxioctl (struct ifnet *ifp,
              int cmd,
              caddr_t data);
```

**Arguments**

- `ifp`
  Specifies a pointer to the ifnet structure for the network device.

- `cmd`
  Specifies an integer ioctl() command value.

- `data`
  Specifies a character pointer to a data buffer.

**Description**

This routine, referenced by the procedure handle if_ioctl, is invoked when an ioctl() command is issued to the network device. The network device driver must handle the ioctl() command values SIOCSIFADDR (set interface address) and SIOCSIFFLAGS (set interface flags), and any others you deem appropriate.

This routine is not called directly by user code.

**Return Values**

- OK if successful.
- An errno value if not successful.
Network Driver Operations

### xxOutput()

Transmit a frame over the network for network device xx

### C Syntax

```c
#include <if.h>
#include <if_ether.h>

int xxOutput (struct ifnet *ifp,
               struct mbuf *m,
               struct sockaddr *dst);
```

### Arguments

- **ifp**
  Specifies a pointer to the ifnet structure for the network device.

- **mbuf**
  Specifies a pointer to the network mbuf chain to be converted into a frame and sent out.

- **dst**
  Specifies a pointer to a structure specifying the socket address of the destination.

### Description

This routine, referenced by the procedure handle if_output, converts a network mbuf chain into a link-level frame and transmits the data over the network.

This routine is not called directly by user code.

### Return Values

- OK if successful.
- An errno value if not successful.
Network Driver Operations

**xxReset()**

Reset network device xx

**C Syntax**

```c
#include <if.h>
#include <if_ether.h>
void xxReset (int unit);
```

**Argument**

*unit*

Specifies the network device unit number (integer value).

**Description**

This routine, referenced by the procedure handle `if_reset`, resets the network device. Typically, it marks the device as inactive, then hardware-resets the device, which is left in a non-functioning state. To use the device again after a reset, the `xxInit()` routine must be invoked.

This routine is not called directly by user code.

**xxWatchdog()**

Perform timer functions for the network device xx driver after a specified interval (optional)

**C Syntax**

```c
#include <if.h>
#include <if_ether.h>
LOCAL void xxWatchdog (int unit);
```

**Argument**

*unit*

Specifies the network device unit number (integer value).
Network Driver Operations

Description
This optional routine, referenced by the procedure handle if_watchdog, is provided if the driver needs to use timer functions. The interval, in seconds, after which the watchdog routine is called is specified by the if_timer field of the ifnet structure for the network device.
This routine is not called directly by user code.

Implementation
None of the network drivers supplied by Digital implement xxWatchdog().
7.4 Network Support Routines

Table 7–3 lists routines and macros that are provided by network libraries or header files for device driver use. Examples of the most commonly used of these — `bcopy_from_mbufs()`, `bcopy_to_mbufs()`, `do_protocol_with_type()`, `ether_attach()`, and `netJobAdd()` — are provided in the DECchip 21040 Ethernet driver listing in Section 7.5.

Macro definitions can be found in header files such as `yourCloneTree/h/if.h` and `yourCloneTree/h/net/mbuf.h`.

Additionally, if you purchased a VxWorks Source Kit, routine definitions can be found in modules such as `yourCloneTree/src/netinet/if_subr.c` and `yourCloneTree/src/netwrs/netLib.c`.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcopy_from_mbufs()</td>
<td>Copy data from a network mbuf chain to a driver buffer</td>
</tr>
<tr>
<td>copy_from_mbufs()</td>
<td></td>
</tr>
<tr>
<td>bcopy_to_mbufs()</td>
<td>Copy data from a driver buffer to a network mbuf chain</td>
</tr>
<tr>
<td>copy_to_mbufs()</td>
<td></td>
</tr>
<tr>
<td>build_cluster()</td>
<td>Encapsulate a passed buffer into an mbuf cluster</td>
</tr>
<tr>
<td>check_trailer()</td>
<td>Decode trailer protocol if present</td>
</tr>
<tr>
<td>do_protocol()</td>
<td>Process input data packet IP and ARP data and check link-level type</td>
</tr>
<tr>
<td>do_protocol_with_type()</td>
<td></td>
</tr>
<tr>
<td>ether_attach()</td>
<td>Make the network device available to the network layer</td>
</tr>
<tr>
<td>ether_output()</td>
<td>Send an output packet contained in a passed mbuf chain</td>
</tr>
<tr>
<td>IF_DEQUEUE()</td>
<td>Dequeue mbufs from the network interface queue</td>
</tr>
<tr>
<td>IF_ENQUEUE()</td>
<td>Queue mbufs onto the network interface queue</td>
</tr>
<tr>
<td>netJobAdd()</td>
<td>Add a task-level interrupt-handler routine to the network task (netTask()) job queue</td>
</tr>
<tr>
<td>set_if_addr()</td>
<td>Set the network device address (SIOCSIFADDR ioctl() request)</td>
</tr>
<tr>
<td>splnet()</td>
<td>Set the network processor level for critical processing by taking the network semaphore</td>
</tr>
<tr>
<td>splimp()</td>
<td>(continued on next page)</td>
</tr>
</tbody>
</table>
Network Support Routines

Table 7–3 (Cont.) Network Support Routines Available for Network Drivers

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>splx()</td>
<td>Restore the network processor level after a critical section</td>
</tr>
</tbody>
</table>

bcopy_from_mbufs(), copy_from_mbufs()

Copy data from a network mbuf chain to a driver buffer

C Syntax

```
#include <mbuf.h>

bcopy_from_mbufs (u_char *buf0, 
                   struct mbuf *m, 
                   int len, 
                   int width);
```

C Syntax

```
#include <mbuf.h>

copy_from_mbufs (char *buf0, 
                 struct mbuf *m, 
                 int len);
```

Arguments

buf0
Specifies a pointer to the driver I/O buffer to contain the output data.

m
Specifies a pointer to the mbuf chain with data to be output.

len
Receives the number of bytes actually transferred.

width
For bcopy_from_mbufs(), specifies the data width, in bytes, to use for memory to memory copying in order to observe any relevant boundary alignment restrictions imposed by the hardware. Possible values are NONE, 1, 2, or 4.
Network Support Routines

Description
These macros, defined in `yourCloneTree/h/net/mbuf.h`, copy data from an mbuf chain into a caller-specified buffer area. The macros are called from a network driver's output routine to copy data from an mbuf chain, as received by the driver, into the driver's own I/O buffers.

Return Values
None; however, the count of bytes copied is returned in the `len` variable.

bcopy_to_mbufs(), copy_to_mbufs()
Copy data from a driver buffer to a network mbuf chain

C Syntax
```c
#include <if.h>
#include <if_ether.h>
#include <if_mbuf.h>

struct mbuf *bcopy_to_mbufs (u_char *buf0, int totlen, int off0, struct ifnet *ifp, int width);
```

C Syntax
```c
#include <if.h>
#include <if_ether.h>
#include <if_mbuf.h>

struct mbuf *copy_to_mbufs (u_char *buf0, int totlen, int off0, struct ifnet *ifp);
```

Arguments

`buf0`
Specifies a pointer to the data buffer to be copied into mbufs.
Network Support Routines

\texttt{totlen}\n\hspace{1em} Specifies the maximum number of bytes to be copied.

\texttt{off0}\n\hspace{1em} Specifies an offset to the trailer header; 0 indicates the trailer protocol is not being used.

\texttt{ifp}\n\hspace{1em} Specifies a pointer to the \texttt{ifnet} structure for the network device.

\texttt{width}\n\hspace{1em} For \texttt{bcopy_to_mbufs()}, specifies the data width, in bytes, to use for memory to memory copying. Possible values are NONE, 1, 2, or 4. This argument is used for processors with special byte alignment requirements.

\textbf{Description}

The \texttt{bcopy_to_mbufs()} routine and the \texttt{copy_to_mbufs()} macro copy a specified amount of data from a driver buffer into a network mbuf chain and return a pointer to the first mbuf. The routines are used in the driver’s receive routines to copy received data so that the data can be passed up to higher network protocols.

The \texttt{copy_to_mbufs()} macro is defined in yourCloneTree/h/net/mbuf.h, and simply calls the \texttt{bcopy_to_mbufs()} routine with a width specification of NONE.

\textbf{Return Values}

A pointer to the created mbuf chain, if successful.

NULL if not successful.

\textbf{build_cluster()}

Encapsulate a passed buffer into an mbuf cluster
Network Support Routines

C Syntax

```c
#include <if.h>
#include <if_ether.h>

struct mbuf *build_cluster (u_char *buf0,
               int tolten,
               struct ifnet *ifp,
               u_char ctyper,
               u_char *pRefCnt,
               FUNCPTRT freeRtn,
               int arg1,
               int arg2,
               int arg3);
```

Arguments

- `buf0` Specifies a pointer to the driver buffer, containing data, to be encapsulated.
- `totlen` Specifies the length of the buffer, in bytes.
- `ifp` Specifies a pointer to the `ifnet` structure for the network device.
- `ctype` Specifies the type of the `mbuf` cluster. Type values can be defined by the driver consistently with the values defined in `yourCloneTree/h/net/mbuf.h`.
- `pRefCnt` Specifies a character pointer to the driver-defined location for the buffer reference count, initialized to zero.
- `freeRtn` Specifies a driver free callback routine to be called when the network is finished with the driver buffer.
- `arg1 ... arg3` Specify arguments one through three to `freeRtn`. 
Network Support Routines

Description
This routine is used to surround a passed buffer with an mbuf header and make it a pseudo-cluster. This is called in a driver receive routine, with a pointer to the IP data, in place of copying data from the driver buffer into mbufs, avoiding the need to copy large amounts of input data across the network. Once encapsulated, the driver buffer is managed by the network, which ultimately calls the specified driver-level callback routine to free the buffer.

Note
This routine uses four bytes immediately preceding the loaned buffer, normally the last two bytes of the source Ethernet address and the Ethernet type field. If this information must be protected, drivers should save it to another location before calling build_cluster().

Return Values
Pointer to an mbuf structure if successful.
NULL if not successful.

check_trailer()
Decode trailer protocol if present

C Syntax
```c
#include <if.h>
#include <if_ether.h>

void check_trailer (FAST struct ether_header *eh,
                   FAST unsigned char *pData,
                   FAST int *pLen,
                   FAST int *pOff,
                   FAST struct ifnet *ifp);
```
Network Support Routines

Arguments

\textit{eh}
Specifies a pointer to the Ethernet header.

\textit{pData}
Specifies a pointer to the data, immediately following \textit{eh}.

\textit{pLen}
Specifies a pointer to an integer that receives the length of the trailer frame, if the protocol is being used; otherwise the location is not modified.

\textit{pOff}
Specifies a pointer to an integer that receives the byte offset to the trailer header, if the protocol is being used; otherwise NULL (0).

\textit{ifp}
Specifies a pointer to the network driver's \textit{ifnet} structure.

Description

This routine processes an Ethernet frame encoded with the trailer protocol, which was used in older BSD implementations of UNIX to copy data more efficiently. (The protocol is not recommended for use with VxWorks.) If the protocol is not in use, the \textit{pOff} variable returns NULL.

do\_protocol(), do\_protocol\_with\_type()

Process input data packet IP and ARP data and check link-level type

C Syntax

\begin{verbatim}
#include <if.h>
#include <if_ether.h>

void do_protocol (FAST struct ether_header *eh,
                  FAST struct mbuf *m,
                  FAST struct arpcom *pArpcom,
                  FAST int len);
\end{verbatim}
Network Support Routines

C Syntax

```c
#include <if.h>
#include <if_ether.h>

void do_protocol_with_type (FAST u_short type,
   FAST struct mbuf *m,
   FAST struct arpcom *pArpcom,
   FAST int len);
```

Arguments

- **eh**
  For `do_protocol()`, specifies a pointer to the Ethernet header.

- **type**
  For `do_protocol_with_type()`, specifies a pointer to the Ethernet link-level type field.

- **m**
  Specifies a pointer to the data, a network mbuf chain.

- **pArpcom**
  Specifies a pointer to the driver arpcom structure.

- **len**
  Specifies the length of the input data in bytes.

Description

These routines check the Ethernet link-level type field and process a specified input mbuf chain based on its type. IP data is processed by enqueueing it in the IP input queue and interrupting the IP handler; ARP data is processed by calling `arpinput()` to handle it appropriately. The routines are called by the driver receive routines to move received data up to the network layer.

These routines should not be called at interrupt level.

Digital recommends that new network drivers use `do_protocol_with_type()` rather than `do_protocol()`.
Network Support Routines

ether_attach()

Make the network device available to the network layer

C Syntax

```c
#include <if.h>
#include <if_ether.h>

void ether_attach (struct ifnet *ifp,
                  int unit,
                  char *name,
                  FUNCPTR initRtn,
                  FUNCPTR ioctlRtn,
                  FUNCPTR outputRtn,
                  FUNCPTR resetRtn);
```

Arguments

- **ifp**
  Specifies a pointer to the ifnet structure for the network device driver.

- **unit**
  Specifies the network device unit number (integer value).

- **name**
  Specifies the character-string name of the network device.

- **initRtn**
  Specifies a pointer to the network driver’s initialization routine.

- **ioctlRtn**
  Specifies a pointer to the network driver’s ioctl() request handling routine.

- **outputRtn**
  Specifies a pointer to the network driver’s output routine.

- **resetRtn**
  Specifies a pointer to the network driver’s reset routine.
Network Support Routines

Description

This routine fills in the ifnet structure for the network device and, to make the driver available to upper network layers, calls the if_attach() routine to add the ifnet structure to the list of available network interfaces.

In addition to copying the passed argument values to the appropriate fields of the ifnet structure, the routine sets the if_mtu field to ETHERMTU, and the if_flags field to IFF_BROADCAST.

Typically this routine is called from the network device driver's attach routine, which is called at system startup time in the {usrConfig | bootConfig} module.

ether_output()

Send an output packet contained in a passed mbuf chain

C Syntax

```c
#include <if.h>
#include <if_ether.h>

int ether_output (struct ifnet *ifp,
                 struct mbuf *m0,
                 struct sockaddr *dst,
                 FUNCPTR startRtn,
                 struct arpcom *pArpcom);
```

Arguments

ifp

Specifies a pointer to the ifnet structure for this network device.

m0

Specifies a pointer to the data, a network mbuf chain, to send out.

dst

Specifies a pointer to the Internet address of the destination, used to obtain the Ethernet address of the destination.

startRtn

Specifies a driver routine that dequeues the specified packet from the driver output queue and transmits the packet across the network.
Network Support Routines

*unit*
Specifies a pointer to the arpcom structure for this device.

**Description**
This routine sends out a protocol packet contained in a specified mbuf chain by queueing the packet in the driver output queue and calling the driver start routine specified by the caller. The start routine specified by the driver must dequeue packets from the driver output queue and perform the device-level work to send out the packet.

Trailer protocol encoding is supported if in use (not recommended with VxWorks); see also the check_trailer() routine.

The ether_attach() routine is called by a network driver’s output routine.

**Return Values**
0 if successful.
An errno value if not successful.

**IF_DEQUEUE()**

Dequeuing mbufs from the driver queue

**C Syntax**

```c
#include <if.h>
#include <if_ether.h>

IF_DEQUEUE (struct ifqueue *ifq,
            struct mbuf *m);
```

**Arguments**

*ifq*
Specifies the driver queue from which an mbuf chain is to be dequeued.

*m*
Receives a pointer to the dequeued mbuf chain, or NULL if the queue is empty.
Network Support Routines

Description
This macro, defined in `yourCloneTree/h/net/if.h`, dequeues an mbuf chain from the specified driver queue. To ensure exclusive access to the queue, the network semaphore should be taken with a call to `splnet()` before calling this macro.

**IF_ENQUEUE()**

Queue mbufs onto the driver queue

**C Syntax**

```c
#include <if.h>
#include <if_ether.h>

IF_ENQUEUE (struct ifqueue *ifq,
              struct mbuf *m);
```

**Arguments**

- `ifq`
  Specifies the driver queue on which a specified mbuf chain is to be enqueued.

- `m`
  Specifies a pointer to the mbuf chain to be enqueued.

**Description**
This macro, defined in `yourCloneTree/h/net/if.h`, enqueues a specified mbuf chain on a specified driver queue. To ensure exclusive access to the queue, you should take the network semaphore with a call to `splnet()` before calling this macro.
Network Support Routines

netJobAdd()

Add a task-level interrupt handler routine to the netTask() job queue

C Syntax

```c
#include <if.h>
#include <if_ether.h>

STATUS netJobAdd (FUNCPTY routine,
    long param1,
    long param2,
    long param3,
    long param4,
    long param5);
```

Arguments

- `routine` Specifies the routine to add to the netTask() queue.
- `param1 ... param5` Specify up to five arguments to be passed to the specified routine.

Description

This routine adds a routine to the job queue of the network task, netTask(). The network task provides task-level processing of network-related routines queued to it. It is called by the interrupt handler of a network driver to perform deferred task-level handling of interrupt processing. This allows the driver's interrupt handler to spend as little time as possible processing at interrupt level, with interrupts locked out, and defer as much processing as possible to task level.

The network task waits on an incoming job queue. When the network driver, or any network module, calls netJobAdd() to add a routine to the job queue, netTask() wakes and processes all jobs on the queue in task context. For each routine on the queue, netTask() calls splnet() to take the spl semaphore, calls the queued routine with the provided arguments, and then calls splx() to release the spl semaphore. The return value from the called routine is ignored.

For more information on the network task, see the VxWorks user documentation, including the reference pages for netLib and netTask().
Network Support Routines

Return Values

OK if successful.
ERROR if not successful.

set_if_addr()

Set the network device address (SIOCSIFADDR ioctl() request)

C Syntax

```c
#include <if.h>
#include <if_ether.h>

int set_if_addr (struct ifnet *ifp,
                 char *data,
                 u_char *enaddr);
```

Arguments

- **ifp**
  Specifies a pointer to the ifnet structure for this network device.

- **data**
  Specifies a pointer to the network address of the device, as passed to the
driver’s ioctl() routine.

- **enaddr**
  Specifies a pointer to the network address (currently not used by the routine).

Description

This routine sets a protocol address associated with the device. It marks the
device as down, calls the driver’s initialization routine, specified in the driver’s
ifnet structure to reinitialize, then manipulates ARP structures to set the
protocol address.

If the driver’s initialization routine returns a non-zero value, set_if_addr() will abort and return the value.

The routine is called from the driver’s ioctl() handling function when it is
handling a request to set the interface address (SIOCSIFADDR).
Network Support Routines

Return Values

0 if successful.
Non-zero if the driver’s initialization routine fails.

splnet(), splimp()

Set the network processor level for critical processing by taking the network semaphore.

C Syntax

```c
#include <if.h>
#include <if_ether.h>
int splimp()
```

C Syntax

```c
#include <if.h>
#include <if_ether.h>
int splnet()
```

Description

These routines, which are functionally identical, take the network mutual-exclusion semaphore, spl, to ensure exclusive access to common network data structures that are shared among tasks. (They do not alter the processor interrupt level, as do their UNIX equivalents.) When the calling task is finished with the shared data structures, it releases the semaphore by calling the splx() routine.

splimp() and splnet() can be considered to mark the beginning of a critical section.

Return Value

The previous processor priority level; supply this value when you call splx() to release the semaphore.
Network Support Routines

splx()

Restore network processor level after a critical section

C Syntax

```c
#include <if.h>
#include <if_ether.h>
void splx (int x);
```

Argument

`x`

Specifies the network processor level to restore — the value returned by a previous call to `splimp()` or `splnet()`.

Description

This routine releases the network mutual-exclusion semaphore, spl, taken by a previous call to the `splimp()` or `splnet()` routine, allowing other tasks access to shared network data structures.

`splx()` can be considered to mark the end of a critical section.
7.5 Sample Network Interface Driver

Example 7–4 provides a listing of the DECchip 21040 Ethernet driver, if_tu.c, which is supported on the EB64, EB66, AXPvme, and AXPpci 33 platforms.

You can also access, on line, the network driver source module yourCloneTree/src/demo/AXPenp/if_enp.c. Additionally, if you have purchased a VxWorks Board Porting Kit or Source Kit, you can refer to the network driver source modules in yourCloneTree/src/drv/netif.

Example 7–4 DECchip 21040 Ethernet Driver Listing

/*if_tu.c - DEC tulip ethernet network interface driver*/

* GENERAL INFORMATION
* This module implements the tulip ethernet network interface driver.
* Standard PCI hardware interface is assumed, though portability is provided
* by the use of the "adapter library", which will determine PCI addresses
* and will perform i/o to tulip CSRs. This driver is designed to support
* multiple tulip units per CPU.
*
* LIMITATIONS
* The twisted pair interface is not supported, only the "aui" interface.
* There may be Medulla specific setup values in the high CSRs.
*
* EXTERNAL INTERFACE
* All initialization is performed within the tuattach() routine and there is
* no separate init() routine. Therefore, in the global interface structure,
* the function pointer to the init() routine is NULL. Accessible routine
* names written in global structure include tuIoctl(), tuOutput(),
* tuReset().
*
* The tulip device is a DMA type of device and typically shares access to
* some region of memory with the CPU. This driver is designed for systems
* that directly share memory between the CPU and the tulip. We assume that
* the shared memory is directly available to the tulip device without any
* software intervention or width restrictions. We also expect any cache
* coherency issues to be resolved in hardware. Alternatively, shared memory
* can be allocated in an uncachable memory region and passed to the driver,
* so that all rings and buffers exist in uncachable space.
*
* There are two user-callable routines: tuattach() and tuFilterSetup().
*
* tuattach() is called to init the device and attach it to the o/s network

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

* protocol stack. It will be called automatically from usrRoot if the *
* system is built with the "INCLUDE_TULIP" macro defined (see config.h).*
* tuFilterSetup() is called to change the ethernet address filtering done by *
* the device. When ethernet frames are received, the destination address is *
* examined by tulip hardware to see if the frame should be accepted. This *
* address filter is initially setup (during tuattach) to pass broadcast and *
* one other address, which is the one gotten by calling sysEnetAddrGet *
* (provided in sysLib.c). The user can subsequently call tuFilterSetup() to *
* change the filter. See function for details. */

#include "vxWorks.h"
#include "stdlib.h"
#include "semLib.h"
#include "ioctl.h"
#include "etherLib.h"
#include "netinet/in_var.h"
#include "net/if_subr.h"
#include "adpLib.h"
#include "drv/if_tu.h"
#define sysEnetAddrGet (int, u_char *)
#if (CPU_FAMILY == ALPHA)
#define arch/alpha/rpb.h
#endif

/* Macros to write and read a tulip csr. "a" is one of {CSR0, ... CSR15} */

#define TU_READ(a,d) adpRead (pDrvCtrl->pAdp, 
                      pDrvCtrl->obSrcAdr, 
                      pDrvCtrl->devAdrs+(a), 
                      pDrvCtrl->adpRWflags, &d)

#define TU_WRITE(a,d) adpWrite(pDrvCtrl->pAdp, 
                        pDrvCtrl->obSrcAdr, 
                        pDrvCtrl->devAdrs+(a), 
                        pDrvCtrl->adpRWflags, d)

/* locals */
DRV_CTRL drvCtrl [TU_UNITS]; /* array of driver control structs */

/* externals */
extern int sysEnetAddrGet (int, u_char *); 
#if (CPU_FAMILY == ALPHA)
extern struct rpb *hwrpb_addr;
#endif

/* forward functions */

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
void tuReset (int);
void tuInt (DRV_CTRL *);
void tuHandleRecvInt (DRV_CTRL *);
int tuOutput (IDR *, MBUF *, SOCK *);
int tuIoctl (IDR *, int, caddr_t);
STATUS tuRestart (int);
BOOL tuConvertDestAddr (IDR *, SOCK *, char *, u_short *, MBUF *);
STATUS tuRingInit (DRV_CTRL *);
STATUS tuRingVerify (DRV_CTRL *);
u_int tuGetHash (u_char *);
void tustatShow (void);
STATUS tuFilterSetup (int, int, TU_MULTI *);

/*******************************************************************************
 * tuattach - publish the interface, and initialize the driver and device
 * The routine publishes the "tu" interface by filling in a network
 * interface record and adding this record to the system list; it allocates
 * and initializes descriptor rings and buffers; it resets the device; it
 * attaches the interrupt vector to tuInt(); then starts the device.
 * Parameters:
 * unit - set to 0 for the first device, 1 for second, etc.
 * device address - PCI (not KSEG) address of the tulip CSRs.
 * interrupt vector - expected to have an adapter ID in high 16 bits and
 *  adapter index in the low 16 bits.
 * memory size - number of bytes to allocate to tulip rings and buffers.
 * memory address - Either specifies an explicit memory region for use by
 *  tulip, or has the constant -1 (or 0) to indicate that the
 *  driver should attempt to allocate the shared memory from
 *  the system space.
 * adpRWflags - Platform specific value to be passed, as is, as "flags"
 *  parameter to adpRead/Write.
 * tuObAdrSpace - Platform specific value to be passed, as is, as "address
 *  space" parameter to adpCreateMapPath, when creating the
 *  outbound mapping.
 * RETURNS: OK or ERROR.
 */
```

(continued on next page)
STATUS tuattach
{
    int unit, /* unit number */
    u_int devAdrs, /* PCI address of tulip base */
    int adpVec, /* ADP interrupt vector for tulip */
    u_long memSize, /* #bytes to allocate */
    char *memAdrs, /* address of memory pool or 0 */
    u_long adpRWflags, /* adpRead/Write "flags" */
    u_long tuObAdrSpace, /* adpCreateMapPath outbound "address space" */
    u_long spare3,
    u_long spare4
{
    DRV_CTRL *pDrvCtrl;
    int i, status, tmp;
    u_long mapGran;

    /* Sanity check the unit number, and see if the specified unit is
     already "attached". */

    if (unit < 0 || unit >= TU_UNITS)
        return (ERROR);

    pDrvCtrl = &drvCtrl[unit];
    if (pDrvCtrl->attached)
        return (OK);
    pDrvCtrl->attached = TRUE;

    /* Allocate or check our memory pool. If we have a real address
     in memStart, zero memory there. This will bus error if we’re
     given a bogus address that is != NULL. Better to force a blow
     up at this point where it can be debugged. */

    if (memSize == 0) {
        printf("tuattach: memSize of 0\n");
        return ERROR;
    }

    if ((memAdrs == (char *) -1) || (!memAdrs)) {
        if ((pDrvCtrl->memStart = calloc (memSize, 1)) == (char *) 0) {
            printf("tuattach: calloc failed\n");
            return ERROR;
        } else {
            pDrvCtrl->memStart = memAdrs;
            bzero (pDrvCtrl->memStart, memSize);
        }
    }
    pDrvCtrl->memEnd = pDrvCtrl->memStart + memSize - 1;

    (continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/* Lookup our adapter from the vector we got passed to us. Create
* an "outbound" mapping from cpu to the tulip adapter so we can
* access tulip registers. Throw away the map object handle since
* we never need to use it again. */

pDrvCtrl->pAdp = adpLookup (adpVec);
pDrvCtrl->obSrcAdr = pDrvCtrl->ibSrcAdr = -1L;
pDrvCtrl->adpVec = adpVec;
pDrvCtrl->devAdrs = devAdrs;
pDrvCtrl->adpRWflags = adpRWflags;

if (adpCreateMapPath
    (NULL, /* src adapter (NULL = cpu) */
     pDrvCtrl->pAdp, /* dest adapter */
     pDrvCtrl->devAdrs, /* dest address to map */
     CSR15, /* requested size of map */
     0, /* node number of requester */
     tuObAdrSpace, /* addr space */
     0, /* default access type */
     &pDrvCtrl->obSrcAdr, /* source bus address of map */
     &pDrvCtrl->obMapObj, /* handle of created map object */
     &pDrvCtrl->obMapSize, /* created map size */
     &pDrvCtrl->obMapGran /* dest bus mapping granularity */
) == ERROR)
{
    printf ("tuattach: outbound adpCreateMapPath failed, errno = 0x%x\n",
            errno);
    return ERROR;
}

/* Adjust the dest address to account for possible s-g. Expect that
* adpCreateMapPath maps dest address in the first map page. */

pDrvCtrl->devAdrs &= (pDrvCtrl->obMapGran - 1);

/* Now create an "inbound" mapping from adapter to main memory for
* tulip to use. */

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

if (adpCreateMapPath
   (pDrvCtrl->pAdp, /* src adapter */
    NULL, /* dest adapter (NULL = main mem) */
    pDrvCtrl->memStart, /* dest address to map */
    memSize, /* requested size of map */
    0, /* node number, unused */
    0, /* default addr space */
    0, /* default access type */
    &pDrvCtrl->ibSrcAdr, /* source bus address of map */
    &pDrvCtrl->ibMapObj, /* handle of created map object */
    &pDrvCtrl->ibMapSize, /* created map size */
    &mapGran /* dest bus mapping granularity */
 ) == ERROR) {
   printf (“tuattach: inbound adpCreateMapPath failed, errno = 0x%x\n”,
          errno);
   return ERROR;
}

/* We need to know where in cpu address space the mapping actually
 * started. For a direct mapped interface it will be KSEG(0).
 */

pDrvCtrl->ibMapStart = (char **) ((u_long) pDrvCtrl->memStart) &
1 - mapGran;

/* Copy our ethernet address into the device block. CSR9 doesn’t seem
 * to work on Medulla – returns 0’s. However it does work on EBxx platforms.
 * So, call out to the BSP to get the ethernet address, if that returns
 * an error, read it from Tulip CSR9.
 */

if (sysEnetAddrGet (unit, (u_char *)pDrvCtrl->idr.ac_enaddr) == ERROR) {
   TU_WRITE (CSR9, 0);
   mb ();
   for (i = 0; i < 6; i++) {
      do {
         TU_READ (CSR9, tmp);
         mb ();
      } while (tmp & 0x80000000);
      pDrvCtrl->idr.ac_enaddr[i] = tmp & 0xff;
   }
}

/* Publish the interface data record */

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

ether_attach (&pDrvCtrl->idr.ac_if, 
    unit, 
    "tu", 
    (FUNCPTTR) NULL, 
    (FUNCPTTR) tuIoctl, 
    (FUNCPTTR) tuOutput, 
    (FUNCPTTR) tuReset);

pDrvCtrl->idr.ac_if.if_flags &= ~(IFF_UP | IFF_RUNNING);
/* Create the transmit semaphore, used for accessing transmit ring * at task level. */

pDrvCtrl->TxSem = semMCreate (SEM_Q_PRIORITY |
    SEM_DELETE_SAFE |
    SEM_INVERSION_SAFE);
if (pDrvCtrl->TxSem == NULL) {
    printf ("tuattach: error creating transmit semaphore\n");
    return (ERROR);
}

pDrvCtrl->RxSem = semMCreate (SEM_Q_PRIORITY |
    SEM_DELETE_SAFE |
    SEM_INVERSION_SAFE);
if (pDrvCtrl->RxSem == NULL) {
    printf ("tuattach: error creating recv semaphore\n");
    return (ERROR);
}

if (tuRestart (unit) == ERROR)
    return ERROR;

/* connect our ISR and enable interrupt at the SIO controller */
if (adpIntConnect (pDrvCtrl->adpVec, (long) pDrvCtrl, tuInt, NULL, NULL) == ERROR) {
    printf ("tuattach: cannot intConnect\n");
    return (ERROR);
}

if (adpSetBusIntEna (pDrvCtrl->pAdp, pDrvCtrl->adpVec, 0) == ERROR) {
    printf ("tuattach: adpSetBusIntEnable failed\n");
    return (ERROR);
}

pDrvCtrl->idr.ac_if.if_flags |= (IFF_UP | IFF_RUNNING | IFF_NOTRAILERS);
return (OK);

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/*******************************************************************************
* tuReset - reset the interface
* Mark interface as inactive & reset the chip
*/
void tuReset (int unit) {
    int csr12, csr13, csr14, csr15;
    DRV_CTRL *pDrvCtrl = & drvCtrl [unit];
    pDrvCtrl->idr.ac_if.if_flags &= ~(IFF_UP | IFF_RUNNING);
    TU_WRITE (CSR0, CSR0_RESET);
    mb ();
    /* need to wait 50 PCI bus cycles = 50 * 30 nS; use 3 uSec */
    sysDelay (3);
    mb ();
    TU_WRITE (CSR12, CSR12_OK);
    /*
    * The following is for the quicksilver board, to
    * set up the TULIP for operation over the thin-wire port
    */
    #if (CPU_FAMILY == ALPHA)
    if (hwrpb_addr->rpb_systype == SYSTYPE_QUICKSILVER) {
        TU_WRITE (CSR13, CSR13_OK_QS);
        TU_WRITE (CSR14, CSR14_OK_QS);
        TU_WRITE (CSR15, CSR15_OK_QS);
    } else {
        TU_WRITE (CSR13, CSR13_OK);
        TU_WRITE (CSR14, CSR14_OK);
        TU_WRITE (CSR15, CSR15_OK);
    }
    #endif
    mb ();
    /* set operating conditions */
    /* 16 longword bursts, naturally aligned, equal tx and rx arbitration */
    TU_WRITE (CSR0, CSR0_16LWALIGN | (16 << CSR0_BURST_SHIFT) | CSR0_BAR);
    mb ();
}

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/*******************************************************************************
* tuRestart – restart the device after having stopped it
* 
* This routine restarts the chip after a nasty error. Stop the tx and rx
* machines, clear all status bits, check all descriptor and buffer pointers,
* check csr3 and csr4, and then start tx and rx machines to restart the tulip.
*/

STATUS tuRestart (int unit) {
  DRV_CTRL *pDrvCtrl;
  u_int tmp;
  TU_MULTI addr [TU_NMULTI + 1];
  extern char etherbroadcastaddr[6];
  pDrvCtrl = &drvCtrl[unit];
  pDrvCtrl->restartCount++;
  tuReset (unit);
  if (pDrvCtrl->restartCount == 1 ||
      tuRingVerify (pDrvCtrl) == ERROR)
    if (tuRingInit (pDrvCtrl) == ERROR) {
      printf("tuRingInit failed.\n");
      return (ERROR);
    }

  /* Write rx ring pointer, tx ring pointer, start tx machine, and send*
   * the setup frame.
   */
  tmp = pDrvCtrl->ibSrcAdr +
    ((u_long) pDrvCtrl->rring & (pDrvCtrl->ibMapSize - 1));
  TU_WRITE (CSR3, tmp);
  tmp = pDrvCtrl->ibSrcAdr +
    ((u_long) pDrvCtrl->tring & (pDrvCtrl->ibMapSize - 1));
  TU_WRITE (CSR4, tmp);
  mb ();
  TU_WRITE (CSR6, CSR6_72TR | CSR6_ST);
  mb ();

  /* Zero the addr array to invalidate all entries, then copy our perfect*
   * address match to one entry and validate it, and copy broadcast address*
   * to the second entry and validate it.
   */

  (continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

bzero ((char *) addr, (TU_NMULTI + 1) * sizeof (TU_MULTI));
bcopy ((char *) pDrvCtrl->ldr.ac_enaddr, (char *) addr[0].dst, 6);
addr[0].valid = 1;
bcopy ((char *) etherbroadcastaddr, (char *) addr[1].dst, 6);
addr[1].valid = 1;

if (tuFilterSetup (unit, TU_16_PERFECT, addr) == ERROR)
    return (ERROR);
/* Now clear all status bits, enable interrupts, and start both the
* rx and tx machines. */
TU_WRITE (CSR5, CSR5_WRITABLE);
mb ();
TU_WRITE (CSR7, CSR5_EB | CSR5_NIS | CSR5_AIS | CSR5_SE | CSR5_RPS |
    CSR5_RU | CSR5_RI | CSR5_UNF | CSR5_TU | CSR5_TPS | CSR5_TI);
mb ();
TU_WRITE (CSR6, CSR6_72TR | CSR6_ST | CSR6_SR);
mb ();
return OK;

/*******************************************************************************
* tuInt - handle tulip device interrupt
* This is the Interrupt Service Routine for tulip. Check for errors;
* report them, clear them, and restart the device. Check for received
* frames and have them processed by tuHandleRecvInt (via netJobAdd).
* Reclaim transmit buffers if tulip is done with them.
*/
void tuInt (DRV_CTRL *pDrvCtrl) {
    TUDESC *txDesc;
    int *pReclaimIndex;
    int *pIndex;
    u_int csr5stat;
    int status;
pDrvCtrl->totInts++;
    /* Read the device status register. If no interrupt exists, return. */
    TU_READ (CSR5, csr5stat);
    if (!(csr5stat & (CSR5_NIS | CSR5_AIS)))
        return;
    /* (continued on next page)*/
/* Clear any interrupt conditions shown in the status we’ve read. 
   * Any writable bits will be write 1 to clear. 
   */

TU_WRITE (CSR5, (CSR5_WRITABLE & csr5stat));
mb ();

/* Handle any errors. */
if (csr5stat & CSR5_AIS) {
    ++pDrvCtrl->csr5Errs;
    ++pDrvCtrl->idr.ac_if.if_ierrors;
    /* print the system (PCI bus) errors */
    if (csr5stat & CSR5_SE) {
        if (csr5stat & CSR5_EB == CSR5_PAR)
            logMsg ("tuInt: PCI parity error\n\r");
        if (csr5stat & CSR5_EB == CSR5_MABORT)
            logMsg ("tuInt: PCI master abort\n\r");
        if (csr5stat & CSR5_EB == CSR5_TABORT)
            logMsg ("tuInt: PCI target abort\n\r");
    }
    /* print the TP errors, which we don’t expect to get */
    if (csr5stat & CSR5_LNF)
        logMsg ("tuInt: link fail??\n\r");
    if (csr5stat & CSR5_AT)
        logMsg ("tuInt: aui/tp pin changed??\n\r");
    if (csr5stat & CSR5_RWT)
        logMsg ("tuInt: rx watchdog timeout??\n\r");
    if (csr5stat & CSR5_TJT)
        logMsg ("tuInt: tx jabber timeout??\n\r");
    /* sum up the expected errors */
    if (csr5stat & CSR5_RU) pDrvCtrl->rxUnavail++;
    /* Queue a job to restart the rx and tx if they have "stopped" 
     * (which is != "suspended"). Mark interface as not UP until 
     * we restart it. */
    if (csr5stat & (CSR5_TPS | CSR5_RPS)) {
        pDrvCtrl->idr.ac_if.if_flags &= ~(IFF_UP | IFF_RUNNING);
        (void) netJobAdd ((FUNCPTR) tuRestart,
            pDrvCtrl->idr.ac_if.if_unit,
            0, 0, 0, 0);
    }
}

(continued on next page)
Example 7–4 (Cont.) DEChip 21040 Ethernet Driver Listing

/* Handle any input packets. Queue a job to process them unless the */
/* the job is already running. */
if ((csr5stat & CSR5_RI) &&
   !(pDrvCtrl->flags & TU_RCV_JOB_ACTIVE)) {
   pDrvCtrl->flags |= TU_RCV_JOB_ACTIVE;
   (void) netJobAdd ((FUNCPTR) tuHandleRecvInt,
                    (u_long) pDrvCtrl,
                    0, 0, 0, 0);
}
/* Handle any output packets. */
if (!(csr5stat & CSR5_TI))
   return;
pReclaimIndex = &pDrvCtrl->reclaimIndex; /* last reclaimed descriptor */
pTindex = &pDrvCtrl->tindex; /* last used descriptor */
while (*pReclaimIndex != *pTindex) {
   txDesc = pDrvCtrl->tring + *pReclaimIndex;
   /* Don’t touch the descriptor if the buffer is still owned by */
   /* TULIP, or if tuOutput is in process of sending (flag will */
   /* be set) and it’s the last descriptor, which will be the one */
   /* being set up in tuOutput. */
   if (txDesc->own || (pDrvCtrl->flags & TU_TX_JOB_ACTIVE) &&
       (txDesc == pDrvCtrl->tring + *pTindex)) break;
}
/* update interface stats */
status = txDesc->status;
pDrvCtrl->idr.ac_if.if_collisions +=
   (status & TU_EC) ? 16:((status & TU_CC) >> TU_CCSHIFT);
/* Check the tx descriptor for error conditions. */
if (status & TU_ES) {
   pDrvCtrl->idr.ac_if.if_oerrors++; /* update interface stats */
   pDrvCtrl->idr.ac_if.if_opackets--;
   /* print out the unexpected errors */

(continued on next page)
Example 7–4 (Cont.) DEChipt 21040 Ethernet Driver Listing

if (status & TU_LO)
    logMsg ("tuInt: lost carrier (possible cable problem)\n"");
if (status & TU_LF)
    logMsg ("tuInt: tp link test fail??\n");
/* sum up expected errors */
if (status & TU_LC && !(status & TU_UF)) pDrvCtrl->txLateColl++;
if (status & TU_EC) pDrvCtrl->txExcessColl++;
if (status & TU_UF) pDrvCtrl->txUflow++;
if (status & TU_DE) pDrvCtrl->txWait++;
}
if (status & TU_LASTRING)
    *pReclaimIndex = 0;
else
    (*pReclaimIndex)++;
}
/* tuHandleRecvInt - task level interrupt service for input packets */
/* This routine is called at task level, queued as a "netJob" by tuInt */
/* (interrupt service routine), to process all received ethernet frames. */
void tuHandleRecvInt (DRV_CTRL *pDrvCtrl) {
    TUDESC    *rxDesc;
    char       *buf;
    MBUF       *pMbuf;
    u_char      *pData;
    int         len;
    u_short     ether_type;
    BOOL        hookAte;
    int         status;
    int         tmp;
    /* Do nothing if if_flags say we are down */
(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
if ((pDrvCtrl->idr.ac_if.if_flags & (IFF_UP | IFF_RUNNING)) !=
    (IFF_UP | IFF_RUNNING))
    return;
if (semTake (pDrvCtrl->RxSem, NO_WAIT) == ERROR)
    return;
handleAgain:
    for (rxDesc = pDrvCtrl->rring + pDrvCtrl->rindex;
        rxDesc->own == 0;
        pDrvCtrl->rindex =
        (rxDesc->ctrl & TU_LASTRING) ? 0 : pDrvCtrl->rindex + 1,
        rxDesc = pDrvCtrl->rring + pDrvCtrl->rindex) {
        /* Check status in the rx descriptor to make sure we have the
         * "last segment" and that no errors have occurred before sending
         * it up the protocol stack.
         */
        status = rxDesc->status;
        if (status & TU_LS && status & TU_FS && !(status & TU_ES)) {
            ++pDrvCtrl->idr.ac_if.if_ipackets;
            /* Get pointer to received packet. We’ll either pass the
             * pointer to a hook routine or copy the data part of the
             * frame into mbufs and pass them up. See etherLib.c for
             * details of etherInputHookRtn.
             */
            buf = pDrvCtrl->ibMapStart +
            (rxDesc->bfaddr - pDrvCtrl->ibSrcAdr);
            len = rxDesc->length;
            if (etherInputHookRtn == NULL ||
                !(*etherInputHookRtn) (&pDrvCtrl->idr.ac_if, buf, len)) {
                len -= SIZEOF_ETHERHEADER;
                pMbuf = bcopy_to_mbufs ((buf + SIZEOF_ETHERHEADER),
                len, 0,
                (IFNET *) & pDrvCtrl->idr.ac_if,
                NONE);
        }
    }
```

(continued on next page)
ether_type = ntohs ((ETH_HDR *) buf)->ether_type);
    if (pMbuf != NULL)
        do_protocol_with_type (ether_type,
                       pMbuf,
                       &pDrvCtrl->idr,
                       len);
    else
        pDrvCtrl->rxNoMbufs++;
    }
} else if (status & TU_LS) {
    pDrvCtrl->idr.ac_if.if_ierrors++;
    if (status & TU_OF) pDrvCtrl->rxOflow++;
    if (status & TU_CE && ! (status & TU_DB)) pDrvCtrl->rxCRCErr++;
    if (status & TU_CS) pDrvCtrl->rxLateColl++;
    if (status & TU_TL) pDrvCtrl->rxTooLong++;
    if (status & TU_RF && ! (status & TU_OF)) pDrvCtrl->rxRunt++;
    if (status & TU_LE) pDrvCtrl->rxWontFit++;
} else {
    /* Not the last segment. Mark a generic error and drop it. */
    pDrvCtrl->idr.ac_if.if_ierrors++;
    }
    /* Update count of missed rx frames. */
    TU_READ (CSR8, tmp);
    pDrvCtrl->missedRxFrames += 0x1ffff & tmp;
    /* Done with descriptor. Give it to the device. */
    mb ();
    rxDesc->own = 1;
    mb ();
}

pDrvCtrl->flags &= ~TU_RCV_JOB_ACTIVE;
mb ();

/* We need to re-check for rcv packets after clearing the
   * RCV_JOB_ACTIVE flag in order to avoid a race here. Otherwise the
   * ISR could add a receive packet, see that the RCV_JOB_ACTIVE flag is
   * set, and decide to exit, leaving the received packet in limbo till
   * some event causes us to see it.
   */
(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
if (rxDesc->own == 0) {
    pDrvCtrl->flags |= TU_RCV_JOB_ACTIVE;
    goto handleAgain;
}
semGive (pDrvCtrl->RxSem);
```

```c
/*******************************************************************************
*  tuOutput - the driver output routine
*  *
*/
int tuOutput (IDR *pIDR, MBUF *pMbuf, SOCK *pDest) {
    char destEnetAddr [6]; /* space for enet addr */
    u_short packetType; /* type field for the packet */
    DRV_CTRL *pDrvCtrl;
    TUDESC *txDesc;
    char *buf;
    int len;
    /* Check ifnet flags. Return if not up and running. */
    if ((pIDR->ac_if.if_flags & (IFF_UP | IFF_RUNNING)) !=
      (IFF_UP | IFF_RUNNING)) return (ENETDOWN);
    /* Attempt to convert socket addr into enet addr and packet type.
     * If ARP resolution of the address is required, the ARP module
     * will call tuOutput() again to transmit the ARP request packet.
     * In that case we’ll "return (OK)" after the convert.
     */
    if (tuConvertDestAddr (pIDR, pDest, destEnetAddr, &packetType, pMbuf)
      == FALSE)
        return (OK); /* OK is correct */
    pDrvCtrl = &drvCtrl[pIDR->ac_if.if_unit];
    /* Take transmit semaphore before accessing transmit ring AND set
     * the flag for use in tuInt.
     */
    semTake (pDrvCtrl->TxSem, WAIT_FOREVER);
    pDrvCtrl->flags |= TU_TX_JOB_ACTIVE;
    /* See if next tx descriptor is available. If not, free mbufs passed in
     * and bail out.
     */
```

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
    txDesc = pDrvCtrl->tring + pDrvCtrl->tindex;
    if (txDesc->own != 0) {
        pDrvCtrl->txUnavail++;
        m_freem (pMbuf);
        pDrvCtrl->flags &= ~ TU_TX_JOB_ACTIVE;
        semGive (pDrvCtrl->TxSem);
        return (ERROR);
    }

    /* Get pointer to transmit buffer */
    buf = pDrvCtrl->ibMapStart + (txDesc->bfaddr - pDrvCtrl->ibSrcAdr);
    /* Fill in the Ethernet header to xmit_buf */
    bcopy (destEnetAddr, buf, 6);
    bcopy ((char *) pIDR->ac_enaddr, &buf[6], 6);
    ((ETH_HDR *)buf)->ether_type = ntohs (packetType);
    /* Copy packet from MBUFs to our xmit buffer. MBUFs are
     * transparently freed. "len" is written by the routine
     * with the number of bytes copied.
     */
    bcopy_from_mbufs (buf + SIZEOF_ETHERHEADER, pMbuf, len, NONE);
    /* Adjust length by header size, and don't send less than legal size. */
    len = max (len + SIZEOF_ETHERHEADER, ETHERSMALL);
    /* setup transmit descriptor; set owner bit to TULIP; do a tx poll
     * demand to start the transmission.
     */
    txDesc->status = 0;
    txDesc->ctrl = (txDesc->ctrl & (TU_TER | TU_TCH)) | TU_IC | TU_TLS | TU_TFS;
    txDesc->bsize = len;
    mb ();
    txDesc->own = 1;
    mb ();
    TU_WRITE (CSR1, 0);
    mb ();

    /* Advance our tx descriptor index. If we're already at the last
     * descriptor in the ring, zero the index.
     */
```

if (txDesc->ctrl & TU_LASTRING)
    pDrvCtrl->tindex = 0;
else
    pDrvCtrl->tindex++;

pIDR->ac_if.if_opackets++; /* update the interface stats */
/* Release "exclusive" access of transmit ring */
pDrvCtrl->flags &= ~TU_TX_JOB_ACTIVE;
semGive (pDrvCtrl->TxSem);
return (OK);
}

/*******************************************************************************

* tuIoctl - the driver I/O control routine
* * Process an ioctl request.
* */
int tuIoctl (IDR *ifp, int cmd, caddr_t data) {
    int error = 0;
    switch (cmd) {
    case SIOCSIFADDR:
        ifp->ac_ipaddr = IA_SIN (data)->sin_addr;
        break;
    case SIOCSIFFLAGS:
        /* No further work to be done */
        break;
    default:
        error = EINVAL;
    }

    return (error);
}

/*******************************************************************************

* tuConvertDestAddr - converts socket addr into enet addr and packet type
* */

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

BOOL tuConvertDestAddr
{
    IDR *pIDR, /* ptr to i/f data record */
    SOCK *pDestSktAddr, /* ptr to a generic sock addr */
    char *pPacketType, /* where to write packet type */
    u_short *pMbuf /* ptr to mbuf data */
}

if (pDestSktAddr->sa_family == AF_INET) {
    struct in_addr destIPAddr;
    int trailers;

    /* Do "internet" family. Define the frame type. Pick off the
     * dest addr. Have arp lookup the IP address from its table
     * or have arp send out packets to get the addresss.
     */
    *pPacketType = ETHERTYPE_IP;
    destIPAddr = ((struct sockaddr_in *) pDestSktAddr)->sin_addr;
    if (!arpresolve (pIDR, pMbuf, &destIPAddr, pDestEnetAddr, &trailers))
        return (FALSE);
    else
        return (TRUE);
}

else if (pDestSktAddr->sa_family == AF_UNSPEC) {
    ETH_HDR *pEnetHdr;

    /* Do "generic" family. Get pointer to ethernet header, get
     * the dest addr and type.
     */
    pEnetHdr = (ETH_HDR *) pDestSktAddr->sa_data;
    bcopy ((char *) pEnetHdr->ether_dhost, pDestEnetAddr, 6);
    *pPacketType = pEnetHdr->ether_type;
    return (TRUE);
}

/* if neither internet or generic, we can't convert it */
return (FALSE);

(continued on next page)
Example 7–4 (Cont.) DEChip 21040 Ethernet Driver Listing

/*******************************************************************************
* tuRingInit - allocate and initialize a list or descriptors and buffers.  
* returns OK, or ERROR if there was a failure.  
* This routine can be called again to reinit the descriptors and buffers,  
* after having first shut down the tulip chip. */
*******************************************************************************/
STATUS tuRingInit (DRV_CTRL *pDrvCtrl) {
    u_long memSize = (u_long) (pDrvCtrl->memEnd - pDrvCtrl->memStart + 1);
    int numBufs;
    char *tmp;
    int i;

    /* From a single chunk of memory, figure out how many descriptor 
    * and buffer pairs will fit. The buffers will all one size, 
    * large enough for a full frame (1518 bytes max). Descriptors 
    * are 4 longwords. First see how many buffers fit, then subtract 
    * out the buffers needed to accomodate the needed descriptors. 
    * The number of descriptors/buffers in the tx ring will be one 
    * quarter of the total; the number of descriptors/buffers in the 
    * rx ring will be the remaining 3/4 of the total. This split 
    * was determined empirically, by examining the status after running. 
    * We'll need to align all descriptors to TUDESC_ALIGN and buffers 
    * to TU_DATABUF_ALIGN. */
    numBufs = memSize / TU_DATABUF_SIZE;
    numBufs -= ((numBufs * sizeof (TUDESC)) / TU_DATABUF_SIZE + 1);
    pDrvCtrl->numTxDesc = numBufs >> 2;
    pDrvCtrl->numRxDesc = numBufs - pDrvCtrl->numTxDesc;
}

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/* With the memory passed to us or malloc’ed by us, assign it like this:
 *--------------------------------------------------------------
 * | Address | Purpose | Size                  |
 * |---------|---------|-----------------------|
 * | memStart | alignment area | < TUDESC_ALIGN |
 * | ->rring | Receive descriptor ring | numRxDesc * sizeof TUDESC |
 * | alignment area | < TUDESC_ALIGN |
 * | ->tring | Transmit descriptor ring | numTxDesc * sizeof TUDESC |
 * | alignment area | < TU_DESC_ALIGN |
 * | ->rbufs | Receive buffers | numRxDesc * ... |
 * | alignment area | < TU_DATABUF_ALIGN |
 * | ->tbufs | Transmit buffers | numTxDesc * TU_DATABUF_SIZE |
 * | memEnd | possible unused area | < 2 * TU_DATABUF_SIZE |
 * | (last location available) |
 *--------------------------------------------------------------*/

pDrvCtrl->rring = (TUDESC *) ROUND_UP (pDrvCtrl->memStart, TUDESC_ALIGN);
pDrvCtrl->tring = (TUDESC *) ROUND_UP (pDrvCtrl->rring + pDrvCtrl->numRxDesc, TUDESC_ALIGN);
pDrvCtrl->rbufs = (TU_DATABUF *) ROUND_UP (pDrvCtrl->tring + pDrvCtrl->numTxDesc, TU_DATABUF_ALIGN);
pDrvCtrl->tbufs = (TU_DATABUF *) ROUND_UP (pDrvCtrl->rbufs + pDrvCtrl->numRxDesc, TU_DATABUF_ALIGN);

/* Reset all indices. */
pDrvCtrl->rindex = pDrvCtrl->tindex = pDrvCtrl->reclaimIndex = 0;

/* Make the descriptors point to the corresponding buffers (using
 PCI bus physical addresses), and write in the buffer size.
 Receive descriptors get owned by tulip now. Also mark the
 last rx and tx descriptors in the list to cause the tulip
 (and us) to close the ring back to the start. */

(continued on next page)
for (i = 0; i < pDrvCtrl->numRxDesc; i++) {
    pDrvCtrl->rring[i].own = 1;
    pDrvCtrl->rring[i].bsize = TU_MAX_FRAMELENGTH;
    pDrvCtrl->rring[i].bfaddr = (u_int) (pDrvCtrl->ibSrcAdr +
       {(u_long) &pDrvCtrl->rbufs[i]} -
       (u_long) pDrvCtrl->ibMapStart);
}
for (i = 0; i < pDrvCtrl->numTxDesc; i++) {
    pDrvCtrl->tring[i].bsize = TU_MAX_FRAMELENGTH;
    pDrvCtrl->tring[i].bfaddr = (u_int) (pDrvCtrl->ibSrcAdr +
       {(u_long) &pDrvCtrl->tbufs[i]} -
       (u_long) pDrvCtrl->ibMapStart);
}
pDrvCtrl->rring[pDrvCtrl->numRxDesc - 1].ctrl = TU_LASTRING;
pDrvCtrl->tring[pDrvCtrl->numTxDesc - 1].ctrl = TU_LASTRING;
return (tuRingVerify (pDrvCtrl));

********************************************************************************
* tuRingVerify - check that the rings and buffers are consistent and
* within bounds.
*
* Make sure that the rx and tx rings
* 1. each start aligned to TUDESC_ALIGN
* 2. each fit in a memory area bounded by memStart and memEnd
* 3. don’t overlap
* 4. have TU_LASTRING marked in only the last ring
*
* For each descriptor in the tx and rx rings, check that bsize and
* bfaddr fields describe the corresponding buffer found by indexing
* into the rbufs and tbufs. Check that bsize2 and bfaddr2 are zero.
*
* Finally, check that the rbufs and tbufs:
* 1. each start aligned to TU_DATABUF_ALIGN
* 2. each fit in a memory area bounded by memStart and memEnd
* 3. don’t overlap
*
* Assume each descriptor in ring abuts the previous one and that
* pDrvCtrl->numTxDesc is the number of descriptors in the tx ring and
* pDrvCtrl->numRxDesc is the number of descriptors in the rx ring. Also
* assume that each descriptor points to one buffer of size
* TU_DATABUF_SIZE.
*
* No assumption is made about where rings or buffers should belong
* in the pool.

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

* Returns OK or ERROR, and prints out reason for the error.
*/

STATUS tuRingVerify (DRV_CTRL *pDrvCtrl) {
    STATUS status;
    int i, tmp;
    char *rxRingEnd, *txRingEnd, *rxBufEnd, *txBufEnd;
    if (pDrvCtrl == 0) /* if NULL, assume tu0 */
        pDrvCtrl = &drvCtrl[0];
    status = OK;
    rxRingEnd = (char *) &pDrvCtrl->rring[pDrvCtrl->numRxDesc - 1] - 1;
    txRingEnd = (char *) &pDrvCtrl->tring[pDrvCtrl->numTxDesc - 1] - 1;
    tmp = 0;
    if ((ULONG) pDrvCtrl->rring & (TUDESC_ALIGN - 1)) tmp = 0x1;
    if ((ULONG) pDrvCtrl->tring & (TUDESC_ALIGN - 1)) tmp = 0x2;
    if ((ULONG) pDrvCtrl->rring < (ULONG) pDrvCtrl->memStart) tmp = 0x4;
    if ((ULONG) rxRingEnd > (ULONG) pDrvCtrl->memEnd) tmp = 0x10;
    if ((ULONG) txRingEnd > (ULONG) pDrvCtrl->memEnd) tmp = 0x20;
    if (tmp) {
        status = ERROR;
        printf(" rx/tx ring alignment or boundary failure; status 0x%x\n",
            tmp);
    }
    if (pDrvCtrl->tring <= pDrvCtrl->rring &&
        txRingEnd > (char *) pDrvCtrl->rring ||
        pDrvCtrl->rring <= pDrvCtrl->tring &&
        rxRingEnd > (char *) pDrvCtrl->tring) {
        printf(" rx overlaps tx ring\n");
        status = ERROR;
    }
    for (i = 0; i < pDrvCtrl->numRxDesc; i++) {
        if (pDrvCtrl->rring[i].own == 1 &&
            (pDrvCtrl->rring[i].bsize != TU_MAX_FRAMELENGTH ||
            pDrvCtrl->rring[i].bsize2 != 0)) {
            printf(" bad buffer sizes at rx descriptor %d (=%p)\n",
                i, &pDrvCtrl->rring[i]);
            status = ERROR;
        }
    }
}

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

    if (pDrvCtrl->rring[i].bfaddr !=
        (pDrvCtrl->ibSrcAdr +
         (u_int) (((char *)pDrvCtrl->rbufs[i]) - pDrvCtrl->ibMapStart)) ||
        pDrvCtrl->rring[i].bfaddr2 != 0) {
        printf("bad buffer addresses at rx descriptor %d (=%p)\n", i, &pDrvCtrl->rring[i]);
        status = ERROR;
    }
    if (i < pDrvCtrl->numRxDesc - 1 &&
        !((pDrvCtrl->rring[i].ctrl & TU_LASTRING)) {  
        printf("TU_LASTRING set before end, rx descriptor %d (=%p)\n", i, &pDrvCtrl->rring[i]);
        status = ERROR;
    }
    if (i == pDrvCtrl->numRxDesc - 1 &&
        !(pDrvCtrl->rring[i].ctrl & TU_LASTRING)) {  
        printf("TU_LASTRING not set in the last rx descriptor %d (=%p)\n", i, &pDrvCtrl->rring[i]);
        status = ERROR;
    }
}

for (i = 0; i < pDrvCtrl->numTxDesc; i++) {
    if (i >= pDrvCtrl->tindex &&
        i <= pDrvCtrl->reclaimIndex &&
        pDrvCtrl->tring[i].own == 1) {
        printf("tx ownership wrong at descriptor %d (=%p)\n", i, &pDrvCtrl->tring[i]);
        status = ERROR;
    }
    if (pDrvCtrl->tring[i].bsize == 0 ||
        pDrvCtrl->tring[i].bsize2 != 0) {
        printf("bad buffer sizes at tx descriptor %d (=%p)\n", i, &pDrvCtrl->tring[i]);
        status = ERROR;
    }
    if (pDrvCtrl->tring[i].bfaddr !=
        (pDrvCtrl->ibSrcAdr +
         (u_int) (((char *)pDrvCtrl->tbufs[i]) - pDrvCtrl->ibMapStart)) ||
        pDrvCtrl->tring[i].bfaddr2 != 0) {
        printf("bad buffer addresses at tx descriptor %d (=%p)\n", i, &pDrvCtrl->tring[i]);
        status = ERROR;
    }
}

(continued on next page)
Example 7–4 (Cont.) DEChip 21040 Ethernet Driver Listing

```c
if (i < pDrvCtrl->numTxDesc - 1 &&
    (pDrvCtrl->tring[i].ctrl & TU_LASTRING)) {
    printf(" TU_LASTRING set before end, tx descriptor %d (=%p)\n", 
           i, &pDrvCtrl->tring[i]);
    status = ERROR;
}
if (i == pDrvCtrl->numTxDesc - 1 &&
    !(pDrvCtrl->tring[i].ctrl & TU_LASTRING)) {
    printf(" TU_LASTRING not set in the last tx descriptor %d (=%p)\n", 
           i, &pDrvCtrl->tring[i]);
    status = ERROR;
}

rxBufEnd = (char *) &pDrvCtrl->rbufs[pDrvCtrl->numRxDesc - 1] - 1;
txBufEnd = (char *) &pDrvCtrl->tbufs[pDrvCtrl->numTxDesc - 1] - 1;

    tmp = 0;
    if ((ULONG) pDrvCtrl->rbufs & (TU_DATABUF.ALIGN - 1)) tmp = 0x1;
    if ((ULONG) pDrvCtrl->tbufs & (TU_DATABUF.ALIGN - 1)) tmp = 0x2;
    if ((ULONG) pDrvCtrl->rbufs < (ULONG) pDrvCtrl->memStart) tmp = 0x4;
    if ((ULONG) rxBufEnd > (ULONG) pDrvCtrl->memEnd) tmp = 0x8;
    if ((ULONG) pDrvCtrl->rbufs < (ULONG) pDrvCtrl->memStart) tmp = 0x10;
    if ((ULONG) rxBufEnd > (ULONG) pDrvCtrl->memEnd) tmp = 0x20;
    if (tmp) {
        printf("tuRingVerify: rx,tx buffer alignment or boundary failure: 0x%x\n", 
               tmp);
        status = ERROR;
    }

    if (pDrvCtrl->rbufs <= pDrvCtrl->rbufs &&
        (ULONG) txBufEnd > (ULONG) pDrvCtrl->rbufs ||
        pDrvCtrl->rbufs <= pDrvCtrl->tbufs &&
        (ULONG) rxBufEnd > (ULONG) pDrvCtrl->tbufs) {
        printf("tuRingVerify: rx buffer overlaps tx buffer\n");
        status = ERROR;
    }
```

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

    printf ("tuRingVerify: rx ring overlaps tx or rx buffer\n");
    status = ERROR;
}

    if ((ULONG) pDrvCtrl->tbufs <= (ULONG) pDrvCtrl->tring &&
        (ULONG) pDrvCtrl->tring <= (ULONG) pDrvCtrl->tbufs &&
        (ULONG) txBufEnd > (ULONG) pDrvCtrl->tring ||
        (ULONG) pDrvCtrl->tring <= (ULONG) pDrvCtrl->tbufs &&
        (ULONG) txRingEnd > (ULONG) pDrvCtrl->tring ||
        (ULONG) pDrvCtrl->rbufs <= (ULONG) pDrvCtrl->tring &&
        (ULONG) rxBufEnd > (ULONG) pDrvCtrl->tring ||
        (ULONG) pDrvCtrl->tring <= (ULONG) pDrvCtrl->rbufs &&
        (ULONG) txRingEnd > (ULONG) pDrvCtrl->rbufs) {
    printf ("tuRingVerify: tx ring overlaps tx or rx buffer\n");
    status = ERROR;
}

    return (status);

/***************************************************************************/
*/
* tuGetHash - calculate a hash index for the setup frame
*/
* (From Appendix C of the 21040 spec.)
*/

u_int tuGetHash (u_char *addr) {
    u_int crc = -1;
    u_int const POLY=0x04c11db6;
    u_int msb;
    u_char byte;
    int i, bit, shift;

    for (i = 0; i < 6; i++) {
        byte = addr [i];
        for (bit = 0; bit < 8; bit++) {
            msb = crc >> 31;
            crc <<= 1;
            if (msb ^ (byte & 1))
                crc = (crc ^ POLY) | 1;
            byte >>= 1;
        }
    }

    for (bit = 23, shift = 8, i = 0; shift >= 0; bit++, shift--)
        i |= ((crc >> bit) & 1) << shift;

    return i;

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/* tuFilterSetup - change the ethernet address filtering */

/* There are three independent filter attributes that can be changed by the */
/* user. These can be changed on a running device, and will either take effect */
/* immediately (in the case of mode changes), or take effect when the frame is */
/* processed in the normal order (for address filter changes). Attributes are: */
/* */
/* 1. "promiscuous" mode, which can be enabled or disabled */
/* 2. "pass all multicast" mode, which can be enabled or disabled */
/* 3. hardware address filter, which has two components: the filter type */
/*    and the filter addresses used. Filter type is one of these classes: */
/*    - accept broadcast and one to 16 addresses, which are perfectly */
/*      filtered */
/*    - accept broadcast, 1 station (physical) address which is perfectly */
/*      filtered, and 512 multicast addresses which are imperfectly */
/*      filtered */
/*    - accept broadcast and 512 station/multicast addresses which are */
/*      imperfectly filtered */
/*    - accept all addresses EXCEPT the one to 16 addresses specified */
/*      ("inverse" filtering) */
/* */
/* Arguments: } unit - specifies which attached tulip device to change */
/* type - User specifies type of filtering: */
/* */
/* Optionally specify one of: */
/* TU_16_PERFECT */
/* TU_INVERSE_PERFECT */
/* TU_1_PERFECT_512_MULTI */
/* TU_512_IMPERFECT */
/* */
/* Optionally specify: */
/* TU_PROMISCUOUS (if this bit is NOT set, promiscuous */
/* mode is terminated) */
/* */
/* Optionally specify: */
/* TU_ALL_MULTI (if this bit is NOT set, "all multi" */
/* mode is terminated) */
/* */
/* (if_tu.h has these macro definitions) */
/* */
/* addr - User specifies which filter addresses to use. The */
/* datatype of "addr" is array of structs of type TU_MULTI. */
/* Array indexes 0 to 512 are used, so the user must define */
/* 513 elements. See if_tu.h for struct definition. */
/* */
/* Put either perfect or multicast addresses in */
/* addr[i].dst[0] thru addr[i].dst[5], and set */

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

Example: To set the filter to accept h/w addr 01:02:03:04:05:06 and all multicast addresses, using the shell:

-> q = malloc (1600)
-> bzero (q, 1600)
-> q[0]=0x0403020100000001
-> q[8]=0x00000605
-> tuFilterSetup(0,0x1|0x20,q)

Returns OK or ERROR

See the tulip data sheet specification for details of setting up the setup frame.

STATUS tuFilterSetup (int unit, int type, TU_MULTI *addr) {
    DRV_CTRL *pDrvCtrl;
    TUDESC *txDesc;
    TUDESC savedTxDesc;
    int hashTable[TU_HASHTABLE_SIZE];
    u_char *cbuf;
    u_int *ibuf, i, ii, tmp, index, noSetupFrame=0;
    int status = OK;
    pDrvCtrl = &drvCtrl[unit];
    if (!pDrvCtrl->attached)
        return ERROR;

    /* We'll create a setup frame in the next transmit buffer. First take the transmit semaphore to make sure tuInt doesn’t reclaim the frame we’re working with. See if next tx descriptor is available. If not, give back the semaphore and taskDelay a tenth of a second. Bail out eventually if we can’t get in. */

    for (i = 0; i < 20; i++) {
        semTake (pDrvCtrl->TxSem, WAIT_FOREVER);
        pDrvCtrl->flags |= TU_TX_JOB_ACTIVE;
        txDesc = pDrvCtrl->tring + pDrvCtrl->tindex;
        if (txDesc->own == 0)
            break;

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
pDrvCtrl->flags &= ~TU_TX_JOB_ACTIVE;
semGive (pDrvCtrl->TxSem);
taskDelay (sysClkRateGet () / 10.);
}
if (txDesc->own != 0) {
    printf ("tuFilterSetup: can’t get a tx descriptor\n");
    return ERROR;
}
/* Get pointer to transmit buffer and clear the buffer. It is useful
* to have both char * and int * pointers to the buf.
*/
cbuf = (u_char *) (pDrvCtrl->ibMapStart + 
    (txDesc->bfaddr - pDrvCtrl->ibSrcAdr));
ibuf = (u_int *) cbuf;
bzero ((char *) cbuf, TU_DATABUF_SIZE);
/* We need to hack the tx descriptor for setup frame. Keep the old
* one to restore after the setup frame is sent.
*/
bcopy ((char *) txDesc, (char *) &savedTxDesc, sizeof (TUDESC));
/* Create setup frame based on the type of frame filtering we want */
if ((type & TU_16_PERFECT) || (type & TU_INVERSE_PERFECT)) {
    if (type & TU_INVERSE_PERFECT) {
        txDesc->ctrl |= TU_FT1;
    }
    txDesc->ctrl |= TU_SET;
    for (ii = i = 0; i < 16; i++) {
        if (addr[i].valid) {
            cbuf[ii+0] = addr[i].dst[0];
            cbuf[ii+1] = addr[i].dst[1];
            cbuf[ii+4] = addr[i].dst[2];
            cbuf[ii+5] = addr[i].dst[3];
            cbuf[ii+8] = addr[i].dst[4];
            cbuf[ii+9] = addr[i].dst[5];
            ii += 12;
        }
    }
    if (ii == 0) {
        printf ("tuFilterSetup: no valid addresses passed in\n");
        noSetupFrame++; 
        status = ERROR;
    }
}(continued on next page)
```
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/* now fill in the rest of the 16 if not all are setup */

while (ii < 192) {
    cbuf[ii+0] = cbuf[0];
    cbuf[ii+1] = cbuf[1];
    cbuf[ii+4] = cbuf[4];
    cbuf[ii+5] = cbuf[5];
    cbuf[ii+8] = cbuf[8];
    cbuf[ii+9] = cbuf[9];
    ii += 12;
}

} else if (type & TU_1_PERFECT_512_MULTI) {
    txDesc->ctrl |= TU_SET | TU_FT0;
    /* put the multicast address filter values into the setup frame */
    for (i = 0; i < TU_NMULTI; i++)
        if (addr[i].valid && (addr[i].dst[6] & 0x80)) {
            index = tuGetHash (addr[i].dst);
            ibuf[index/16] |= 1 << (index % 16);
        }

    /* Now find and write the 1 perfect match, starting at cbuf[156] */
    for (ii = 156, i = 0; i < TU_NMULTI; i++)
        if (addr[i].valid && !(addr[i].dst[6] & 0x80)) {
            cbuf[ii+0] = addr[i].dst[0];
            cbuf[ii+1] = addr[i].dst[1];
            cbuf[ii+4] = addr[i].dst[4];
            cbuf[ii+5] = addr[i].dst[5];
            cbuf[ii+8] = addr[i].dst[8];
            cbuf[ii+9] = addr[i].dst[9];
            break;
        }
}

} else if (type & TU_512_IMPERFECT) {
    txDesc->ctrl |= TU_SET | TU_FT1 | TU_FT0;
    /* put the address filter values into the setup frame */

    (continued on next page)
for (i = 0; i < TU_NMULTI; i++) {
    if (addr[i].valid) {
        index = tuGetHash(addr[i].dst);
        ibuf[index/16] |= 1 << (index % 16);
    }
} else {
    noSetupFrame++;
}

if (type & TU_PROMISCUOUS) {
    TU_READ (CSR6, tmp);
    TU_WRITE (CSR6, tmp | CSR6_PR);
    pDrvCtrl->idr.ac_if.if_flags |= IFF_PROMISC;
} else {
    TU_READ (CSR6, tmp);
    TU_WRITE (CSR6, tmp & ~CSR6_PR);
    pDrvCtrl->idr.ac_if.if_flags &= ~IFF_PROMISC;
}

if (type & TU_ALL_MULTI) {
    TU_READ (CSR6, tmp);
    TU_WRITE (CSR6, tmp | CSR6_PM);
    pDrvCtrl->idr.ac_if.if_flags |= IFF_ALLMULTI;
} else {
    TU_READ (CSR6, tmp);
    TU_WRITE (CSR6, tmp & ~CSR6_PM);
    pDrvCtrl->idr.ac_if.if_flags &= ~IFF_ALLMULTI;
}

/* If an address filter was specified, set owner bit to TULIP, *
   * do a tx poll demand (csr1) to start transmission. *
   */
#if 0
printf ("\nsetup frame is:\n");
for (i = 0; i < SETUPFRAME_SIZE; i += 8) {
    printf ("%03d: %02x %02x %02x %02x %03d: %02x %02x %02x %02x\n",
            i, cbuf[i+0], cbuf[i+1], cbuf[i+2], cbuf[i+3],
            i+4, cbuf[i+4], cbuf[i+5], cbuf[i+6], cbuf[i+7]);
}
#endif
if (!noSetupFrame) {

(continued on next page)
Example 7–4 (Cont.)  DECchip 21040 Ethernet Driver Listing

```c
    txDesc->status = 0;
    txDesc->ctrl &= ~(TU_IC | TU_TLS | TU_TFS);
    txDesc->bsize = SETUPFRAME_SIZE;
    mb ();
    txDesc->own = 1;
    mb ();
    TU_WRITE (CSR1, 0);
    mb ();
    /* Wait for the setup frame to be sent, but not forever.
     * Restore the transmit descriptor.
     */
    for (i = 0; i < 20; i++) {
        taskDelay (sysClkRateGet () / 10);
        if (txDesc->own == 0)
            break;
    }
    if (txDesc->own) {
        printf ("ntulip setup frame not sent in 2 sec, giving up.\n");
        status = ERROR;
    }
    bcopy ((char *) &savedTxDesc, (char *) txDesc, sizeof (TUDESC));
    mb ();
    if (!noSetupFrame) {
        /* Advance our tx descriptor index; wrap if we're at the end. */
        if (txDesc->ctrl & TU_LASTRING)
            pDrvCtrl->tindex = 0;
        else
            pDrvCtrl->tindex++;
    }
    pDrvCtrl->flags &= ~TU_TX_JOB_ACTIVE;
    semGive (pDrvCtrl->TxSem);
    mb ();
    return (status);
```

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

/***************************************************************************
*
* tustatShow - print status summary
*
*/
void tustatShow (void) {
  int tmp, tmp1, tmp2, tmp3;int unit;DRV_CTRL *pDrvCtrl;
  for (unit = 0; unit < TU_UNITS; unit++) {
    pDrvCtrl = &drvCtrl[unit];if (pDrvCtrl->attached) {
      printf ("tulip unit %d: \n", unit);
      printf (" attached at pDrvCtrl = %p\n", &pDrvCtrl);
      printf (" CSRs at PCI addr 0x%08x", 
              pDrvCtrl->devAdrs);
      printf (" main mem at PCI addr 0x%08x\n", 
              pDrvCtrl->ibSrcAdr);
      printf ("\n");
      printf (" main mem allocation: %p to %p\n", 
              pDrvCtrl->memStart, pDrvCtrl->memEnd);
      printf (" %d rx descriptors; %d tx descriptors\n", 
              pDrvCtrl->numRxDesc, pDrvCtrl->numTxDesc);
      printf (" rx ring from %p to %p\n", 
              pDrvCtrl->rring, 
              (char *)(pDrvCtrl->rring + pDrvCtrl->numRxDesc) - 1);
      printf (" tx ring from %p to %p\n", 
              pDrvCtrl->tring, 
              (char *)(pDrvCtrl->tring + pDrvCtrl->numTxDesc) - 1);
      printf (" rx bufs from %p to %p\n", 
              pDrvCtrl->rbufs, 
              (char *)(pDrvCtrl->rbufs + pDrvCtrl->numRxDesc) - 1);
      printf (" tx bufs from %p to %p\n", 
              pDrvCtrl->tbufs, 
              (char *)(pDrvCtrl->tbufs + pDrvCtrl->numTxDesc) - 1);
      printf ("\n");
      printf (" rx index= %d; tx index= %d; reclaimIndex= %d\n", 
              pDrvCtrl->rindex, pDrvCtrl->tindex, pDrvCtrl->reclaimIndex);
      printf ("\n");
      TU_READ (CSR0, tmp1);
      TU_READ (CSR3, tmp2);
      TU_READ (CSR4, tmp3);
      printf (" CSR0 = %08x, CSR3 (rx ring) = %08x, CSR4 (tx ring)= %08x\n", 
              tmp1, tmp2, tmp3);
    }
  }
}(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

TU_READ (CSR5, tmp);
printf (" CSR5 = %08x\n", tmp);
/* we shouldn’t expect to see these bits; cleared in ISR */
if (tmp & CSR5_EB)
  printf (" PCI bus error %d\n", (tmp>>23)&7);
if (tmp & CSR5_PAR)
  printf (" = parity error\n");
if (tmp & CSR5_MABORT)
  printf (" = master abort\n");
if (tmp & CSR5_TABORT)
  printf (" = target abort\n");
printf (" tx machine state %d ", (tmp>>20)&7);
switch ((tmp>>20)&7) {
  case 0: printf ("= stopped\n"); break;
  case 1: printf ("= running; fetching tx desc\n"); break;
  case 2: printf ("= running; awaiting EOT\n"); break;
  case 3: printf ("= running; DMA tx buf to FIFO\n"); break;
  case 4: printf ("= ???reserved???\n"); break;
  case 5: printf ("= running; setup pkt\n"); break;
  case 6: printf ("= suspended; underflow or unavail desc\n"); break;
  case 7: printf ("= running; closing tx desc\n"); break;
}
printf (" rx machine state %d ", (tmp>>17)&7);
switch ((tmp>>17)&7) {
  case 0: printf ("= stopped\n"); break;
  case 1: printf ("= running; fetching rx desc\n"); break;
  case 2: printf ("= running; awaiting EOT\n"); break;
  case 3: printf ("= running; awaiting pkt\n"); break;
  case 4: printf ("= suspended; unavailable rx buf\n"); break;
  case 5: printf ("= running; closing rx desc\n"); break;
  case 6: printf ("= running; flushing FIFO due to unavail buf\n"); break;
  case 7: printf ("= running; DMA FIFO to rx buf\n"); break;
}
if (tmp & CSR5_NIS)
  printf (" normal interrupt\n");
if (tmp & CSR5_AIS)
  printf (" error interrupt\n");
if (tmp & CSR5_SE)
  printf (" system error\n");
if (tmp & CSR5_LNF)
  printf (" link fail\n");
if (tmp & CSR5_FD)
  printf (" fullDuplex short frame\n");
if (tmp & CSR5_AT)
  printf (" aui/tp pin changed\n");
if (tmp & CSR5_RWT)
  printf (" rx watchdog timeout\n");

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
if (tmp & CSR5_RPS) 
  printf (" rx machine stopped\n");
if (tmp & CSR5_RU) 
  printf (" rx buffer unavailable\n");
if (tmp & CSR5_RI) 
  printf (" rx interrupt\n");
if (tmp & CSR5_UNF) 
  printf (" tx underflow\n");
if (tmp & CSR5_TJT) 
  printf (" tx jabber timeout\n");
if (tmp & CSR5_TPS) 
  printf (" tx buffer unavailable\n");
if (tmp & CSR5_TI) 
  printf (" tx machine stopped\n");
```

TU_READ (CSR6, tmp1);
TU_READ (CSR7, tmp2);
TU_READ (CSR8, tmp3);
printf (" CSR6 = %08x, CSR7 = %08x, CSR8 = %08x\n", tmp1, tmp2, tmp3);
TU_READ (CSR9, tmp1);
TU_READ (CSR10, tmp2);
TU_READ (CSR11, tmp3);
printf (" CSR9 = %08x, CSR10= %08x, CSR11= %08x\n", tmp1, tmp2, tmp3);
TU_READ (CSR12, tmp1);
TU_READ (CSR13, tmp2);
TU_READ (CSR14, tmp3);
TU_READ (CSR15, tmp);
printf (" CSR12= %08x, CSR13= %08x, CSR14= %08x, CSR15= %08x\n",
  tmp1, tmp2, tmp3, tmp);

(continued on next page)
Example 7–4 (Cont.) DECchip 21040 Ethernet Driver Listing

```c
printf(" total csr5 errors = %lu", pDrvCtrl->csr5Errs);
printf(" rx missed frames = %lu", pDrvCtrl->missedRxFrames);
printf(" rx overflow frames = %lu", pDrvCtrl->rxOflow);
printf(" rx CRC errors = %lu", pDrvCtrl->rxCRCErr);
printf(" rx late collisions = %lu", pDrvCtrl->rxLateColl);
printf(" rx frames too long = %lu", pDrvCtrl->rxTooLong);
printf(" rx runt frames = %lu", pDrvCtrl->rxRunt);
printf(" rx didn’t fit in rx buf = %lu", pDrvCtrl->rxWontFit);
printf(" rx ring full = %lu", pDrvCtrl->rxUnavail);
printf(" rx no mbufs = %lu", pDrvCtrl->rxNoMbufs);
printf(" tx ring full = %lu", pDrvCtrl->txUnavail);
printf(" tx FIFO underflow = %lu", pDrvCtrl->txUflow);
printf(" tx late collisions = %lu", pDrvCtrl->txLateColl);
printf(" tx abort due to >16 collisions = %lu", pDrvCtrl->txExcessColl);
printf(" tx defer sending because carrier asserted = %lu", pDrvCtrl->txWait);
printf(" total interrupts = %lu", pDrvCtrl->totInts);
printf(" total restarts = %lu", pDrvCtrl->restartCount);
```
This chapter explains how to write a device driver for a SCSI controller that is not already supported by VxWorks for Alpha. A SCSI device driver provides low-level functions that are used by the \texttt{scsiLib} routine library, which implements the IEEE SCSI (Small Computer System Interface) Bus Specification.

If you are porting VxWorks to a board containing a SCSI controller chip, first determine if VxWorks already supports a driver for the SCSI chip on the board. If your chip is supported, you do not need to write a new driver; rather, you modify board-specific \texttt{sysScsi} routines that handle board-dependent aspects of the SCSI chip’s operation. For example, you modify \texttt{sysScsiInit()} to provide any necessary board-dependent hardware and software initialization for a SCSI device. In most cases, the existing SCSI driver should require very limited, or no, modification for your target configuration.

If VxWorks does not support a driver for your SCSI chip, you write a driver using this chapter, a SCSI driver template, and BSP source files.

Writing a SCSI device driver includes the following steps:

1. Establish a SCSI driver template, using the driver listings in Section 8.5. If you purchased a VxWorks Board Porting Kit or Source Kit, you can additionally refer to the SCSI driver source modules in yourCloneTree/src/drv/scsi.

2. Select a driver mnemonic for your device, to be used in the driver name, the names of required driver routines, and driver definitions. For example, the NCR 53C90 Advanced SCSI Controller (ASC) SCSI driver uses \texttt{ncr5390} (\texttt{ncr5390Lib.c}, \texttt{ncr5390CtrlCreate()}, \texttt{NCR_5390_SCSI_CTRL}). The NCR 53C810 SCSI I/O Processor (PSIOP) driver driver uses \texttt{ncr810} (\texttt{ncr810Lib.c}, \texttt{ncr810CtrlCreate()}, \texttt{NCR810_SCSI_CTRL}).

3. Code the well-defined set of device-specific driver routines expected by \texttt{scsiLib}, described in Section 8.3.
4. As part of coding required driver routines, you will need to define
device-specific data structures, described in Section 8.2, in a header file
named after your driver. (For example, ncr5390.h or ncr810.h.)

5. Place device address and vector definitions needed by initialization and
driver routines in your BSP's config.h file, or in the target.h file included
by config.h.

6. Check that SCSI support is enabled in config.h. There are four
components to SCSI support:
   - INCLUDE_SCSI — SCSI interface support
   - INCLUDE_SCSI_DMA — SCSI DMA support
   - SCSI_AUTO_CONFIG — SCSI automatic configuration support
   - INCLUDE_SCSI_BOOT — SCSI booting support

7. If you want your driver to be readily portable across multiple platforms
using the same SCSI controller chip, you should code BSP hook routines
to handle board dependencies. During system initialization, the routine
sysScsiInit() is called to perform any board-specific hardware or
software initialization related to the SCSI chip. (You are required to
provide a sysScsiInit() routine if your board supports an onboard
SCSI controller.) The driver may call other BSP routines as needed; for
example, sysWbFlush() is commonly called to flush write buffers (ensuring
previously-issued memory writes finished). In addition to these routines,
you should consider whether adding callouts from the driver to your BSP
would promote the board-independence of your driver code.

8. If you did not enable SCSI automatic configuration (SCSI_AUTO_CONFIG) in config.h, you must edit the SCSI initialization routine,
usrScsiConfig(), in module yourCloneTree/src/config/usrScsi.c, to
reflect the configuration of physical devices on the SCSI bus (ID and logical
unit number).

This chapter addresses the following topics:
   - Driver overview, Section 8.1
   - Driver data structures, Section 8.2
   - Driver operations, Section 8.3
   - Driver support macros, Section 8.4
8.1 Driver Overview

SCSI controllers are used to control various forms of mass-storage devices, including disk and tape drives. The scsiLib library provides a platform-independent interface to SCSI devices, allowing the use of UNIX style I/O and any of the VxWorks file systems (DOS, RT–11, raw) on those devices.

Figure 8–1 shows how SCSI and non-SCSI block device drivers fit into the VxWorks I/O system hierarchy. SCSI devices are block devices that contain file systems, represented in the right half of the diagram. For SCSI devices, the scsiLib library acts as intermediary between the file system and the device driver. scsiLib accepts block device I/O requests, such as read, write, reset, and ioctl(), and issues requests to the device driver for the few low-level, device-dependent functions it requires to execute the SCSI protocol. (scsiLib is described in the VxWorks reference material, including the scsiLib reference page; the block device interface is explained in the VxWorks Programmer’s Guide.)

A SCSI device driver is expected to provide the SCSI_CTRL data structure, described in Section 8.2, and a well-defined set of device-specific routines, described in Section 8.3, for scsiLib to access.

In addition, to allow for any target-specific differences in driver initialization, a SCSI initialization routine, sysScsiInit(), must be added to the system-dependent routines library, sysLib. Any other functions that must account for board-specific differences, such as DMA transfer control, should be implemented in sysLib.c as well.
You should modify the processor-specific definitions file, `yourCloneTree/config/target/config.h`, to include the appropriate definitions for your onboard SCSI device’s I/O addresses, interrupt vectors, and other information needed by your driver. Typically SCSI initialization routines and driver routines use these definitions.

As described in the I/O System chapter of the VxWorks Programmer’s Guide, you include the SCSI interface in the VxWorks system image by defining `INCLUDE_SCSI` in `yourCloneTree/config/target/config.h`. This causes SCSI initialization code to be compiled into the `usrRoot` routine in `usrConfig.c`, which runs during system startup. In particular, `usrRoot()` calls the SCSI initialization routines `sysScsiInit()` and `usrScsiConfig()`.

You can additionally enable SCSI DMA, SCSI automatic configuration, and SCSI booting by defining `INCLUDE_SCSI_DMA`, `SCSI_AUTO_CONFIG`, and `INCLUDE_SCSI_BOOT` in `config.h`.
The routine `sysScsiInit()` , which you provide in `sysLib.c` , initializes the SCSI controller and sets up interrupt handling.

The routine `usrScsiConfig()` , which is found in `yourCloneTree/src/config/usrScsi.c` , configures the SCSI interface. If you did not enable SCSI automatic configuration in `config.h` , you should modify the routine `usrScsiConfig()` to reflect the actual configuration of physical devices on your SCSI bus (device IDs and logical unit numbers).

The routine source code contains an example of the calling sequence to declare a hypothetical configuration, including the definition of physical devices with `scsiPhysDevCreate()` , the creation of logical partitions with `scsiBlkDevCreate()` , and the specification of a file system with either `dosFsDevInit()` or `rt11FsDevInit()` . For more information on the routines used in `usrScsiConfig()` , see the reference pages for `scsiPhysDevCreate()` , `scsiBlkDevCreate()` , `dosFsDevInit()` , `rt11FsDevInit()` , `dosFsMkfs()` , and `rt11FsMkfs()` .

If problems with your configuration occur, you can obtain further information about SCSI bus activity for debugging by setting one of the following variables to TRUE at the beginning of `usrScsiConfig()` :

```c
scsiDebug = TRUE;
scsiIntsDebug = TRUE;
```

`scsiDebug` and `scsiIntsDebug` are global variables defined in `scsiLib` that should not be declared locally. They may be set or reset from the VxWorks shell. Other information related to SCSI driver debugging is provided in Section 8.4.

The `scsiLib` routine library provides routines for configuring SCSI devices in `usrConfig.c` , and for obtaining SCSI physical device information. Table 8–1 lists those routines; for more information about these, and other `scsiLib` routines not listed, see the reference page for `scsiLib` .
Table 8–1  scsiLib Routines Subset for SCSI Device Configuration and Display

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsiAutoConfig()</td>
<td>Configure all devices connected to a SCSI controller</td>
</tr>
<tr>
<td>scsiBlkDevCreate()</td>
<td>Define a logical partition on a SCSI block device</td>
</tr>
<tr>
<td>scsiBlkDevInit()</td>
<td>Initialize fields in a SCSI logical partition</td>
</tr>
<tr>
<td>scsiPhysDevCreate()</td>
<td>Create a SCSI physical device structure</td>
</tr>
<tr>
<td>scsiPhysDevDelete()</td>
<td>Delete a SCSI physical device structure</td>
</tr>
<tr>
<td>scsiPhysDevIdGet()</td>
<td>Return a pointer to SCSI physical device structure</td>
</tr>
<tr>
<td>scsiShow()</td>
<td>List the physical devices attached to a SCSI controller</td>
</tr>
</tbody>
</table>

Table 8–2 shows the initialization and runtime flow of control of a SCSI device driver.

Table 8–2  SCSI Device Driver Flow of Control

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>usrRoot()</strong> =&gt; sysScsiInit()</td>
</tr>
<tr>
<td>Performs board-specific SCSI initialization</td>
</tr>
<tr>
<td>=&gt; xxxCtrlCreate()</td>
</tr>
<tr>
<td>=&gt; intConnect() or adpIntConnect()</td>
</tr>
<tr>
<td>=&gt; xxxCtrlInit()</td>
</tr>
<tr>
<td><strong>usrRoot()</strong> =&gt; usrScsiConfig()</td>
</tr>
<tr>
<td>=&gt; p = scsiPhysDevCreate()</td>
</tr>
<tr>
<td>=&gt; p1 = scsiBlkDevCreate(p)</td>
</tr>
<tr>
<td>=&gt; p2 = scsiBlkDevCreate(p)</td>
</tr>
<tr>
<td>=&gt; dosFsDevInit(p1)</td>
</tr>
<tr>
<td>=&gt; rt11FsDevInit(p2)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

(sysLib.c)

(intArchLib or adpLib)

(sysLib.c (driver))

(continued on next page)
Table 8–2 (Cont.) SCSI Device Driver Flow of Control

<table>
<thead>
<tr>
<th>Runtime</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>scsiLib</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xxxBytesIn(), other driver routines</td>
<td>xxxLib.c (driver)</td>
</tr>
<tr>
<td></td>
<td>Handle device dependencies, call BSP as necessary for board dependencies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>==&gt; sysScsi routines</td>
<td>sysLib.c</td>
</tr>
<tr>
<td>Device</td>
<td>xxxIntr() ISR</td>
<td>xxxLib.c (driver)</td>
</tr>
<tr>
<td></td>
<td>Handles device interrupts</td>
<td></td>
</tr>
</tbody>
</table>

8.2 Driver Data Structures

Each SCSI device driver defines the data structure xxx_SCSI_CTRL, where xxx is a unique mnemonic for the SCSI driver. The structure contains driver-specific information used by scsiLib and the driver itself, such as register information and a status code variable. The xxx_SCSI_CTRL structure contains other important driver data structures, including the SCSI_CTRL controller structure and an array of SCSI_PHYS_DEV structures, one for each SCSI device on the bus.

You can define the xxx_SCSI_CTRL structure according to your configuration and your driver’s needs. The definition should be placed in a header file you establish for the SCSI controller; for examples, see similar SCSI header files provided by Digital in yourCloneTree/h/drv/scsi, such as ncr5390.h for the NCR 53C90 ASC and ncr810.h for the NCR 53C810 PSIO.

Example 8–1 shows the NCR_5390_SCSI_CTRL structure defined by the NCR 53C90 ASC SCSI driver, ncr5390Lib, in ncr5390.h; and Example 8–2 shows the NCR810_SCSI_CTRL structure defined by the NCR 53C810 PSIO SCSI driver, ncr810Lib, in ncr810.h. Some related value and type definitions are omitted from these example, but are available in the header file.
Example 8–1 ncr5390Lib SCSI Driver Data Structure NCR_5390_SCSI_CTRL

typedef struct /* NCR_5390_SCSI_CTRL - NCR 5390 SCSI controller info */
{
    SCSI_CTRL scsiCtrl; /* generic SCSI controller info */
    SCSI_PHYS_DEV *pDevToSelect; /* device to select at intr. level or NULL */
    int devType; /* type of device (see define’s below) */
    TBOOL slowCubeMode; /* TRUE to select slow cable mode */
    TBOOL resetReportDsl; /* TRUE to disable SCSI bus reset reporting */
    TBOOL parityTestMode; /* TRUE to enable parity checking (DO NOT USE!) */
    TBOOL parityCheckEnbl; /* TRUE to enable parity checking */
    UINT8 defaultSelTimeOut; /* default dev. select time-out (units var.) */
    UINT8 clkCvtFactor; /* value of the clock conversion factor */
    UINT8 savedStatReg; /* status register from last interrupt */
    UINT8 savedIntrReg; /* interrupt register from last interrupt */
    UINT8 savedStepReg; /* sequence step register from last interrupt */
    #ifdef HOST_DEC
    volatile unsigned char *pTclReg; /* ptr xfer count LSB reg */
    volatile unsigned char *pTchReg; /* ptr xfer count MSB reg */
    volatile unsigned char *pFifoReg; /* ptr FIFO reg */
    volatile unsigned char *pCmdReg; /* ptr command reg */
    volatile unsigned char *pStatReg; /* ptr status reg */
    volatile unsigned char *pIntrReg; /* ptr interrupt reg */
    volatile unsigned char *pStepReg; /* ptr sequence step reg */
    volatile unsigned char *pFlgsReg; /* ptr FIFO flags reg */
    volatile unsigned char *pCfg1Reg; /* ptr configuration 1 reg */
    volatile unsigned char *pClkReg; /* ptr clock conversion factor reg */
    volatile unsigned char *pTestReg; /* ptr test mode reg */
    volatile unsigned char *pCfg2Reg; /* ptr configuration 2 reg */
    volatile unsigned char *pCfg3Reg; /* ptr configuration 3 reg */
    #else /* HOST_DEC */
    volatile UINT8 *pTclReg; /* ptr xfer count LSB reg */
    volatile UINT8 *pTchReg; /* ptr xfer count MSB reg */
    volatile UINT8 *pFifoReg; /* ptr FIFO reg */
    volatile UINT8 *pCmdReg; /* ptr command reg */
    volatile UINT8 *pStatReg; /* ptr status reg */
    volatile UINT8 *pIntrReg; /* ptr interrupt reg */
    volatile UINT8 *pStepReg; /* ptr sequence step reg */
    volatile UINT8 *pFlgsReg; /* ptr FIFO flags reg */
    volatile UINT8 *pCfg1Reg; /* ptr configuration 1 reg */
    volatile UINT8 *pClkReg; /* ptr clock conversion factor reg */
    volatile UINT8 *pTestReg; /* ptr test mode reg */
    volatile UINT8 *pCfg2Reg; /* ptr configuration 2 reg */
    volatile UINT8 *pCfg3Reg; /* ptr configuration 3 reg */
    #endif /* HOST_DEC */
} NCR_5390_SCSI_CTRL;
typedef struct /* NCR_810_SCSI_CTRL - NCR810 */ /* SCSI controller info */
{
    SCSI_CTRL scsiCtrl; /* generic SCSI controller info */
    SEM_ID pMutexData; /* use to protect global siop data */
    SCSI_PHYS_DEV *pDevToSelect; /* device to select at intr. level or NULL */
        /* Hardware implementation dependencies */
    TBOOL slowCableMode; /* TRUE to select slow cable mode */
    int chipType; /* Device type NCR8X0_TYPE */
    /* Only 810 Supported today */
    int devType; /* type of device (see define’s below) */
    UINT8 ctrlIdPow2; /* controller id in power of 2 (1000000=id 7) */
    TBOOL resetReportDsb1; /* TRUE to disable SCSI bus reset reporting */
    TBOOL parityTestMode; /* TRUE enable parity test mode (DO NOT USE) */
    TBOOL parityCheckEnbl; /* TRUE to enable parity checking */
    UINT8 defaultSelTimeOut; /* default dev. select time-out (units var.) */
    UINT8 clkCvtFactor; /* value of the clock conversion factor */
    UINT8 saveIstat; /* Save Reg under interrupt */
    UINT8 saveSist0; /* Save Reg under interrupt */
    UINT8 saveDstat; /* Save Reg under interrupt */
    UINT8 saveIdentMsg; /* Save Indent message in at interrupt level */
        /* when reconnect. */
    TBOOL commandRequest; /* To check if we are reconnected with a */
        /* request pending */
    UINT saveScriptIntrStat; /* Save intr script status under interrupt */
    SEMAPHORE singleStepSem; /* use to debug script in single step mode */
    NCR_CTL_CTX *pNcrCtxt; /* pointer to Array of data shared by script */
        /* and driver */
    NCR_CTL *pNcrCtl; /* Current context pointer for intr level */
    NCR_CTL *pNcrCtlCmd; /* current ctxt pointer to the command */
        /* pending */
    NCR810_HW_REGS hwRegs; /* values used for hardware dependent regs - */
        /* not in any particular order */

(continued on next page)
Example 8–2 (Cont.) ncr810Lib SCSI Driver Data Structure NCR810_SCSI_CTRL

volatile UINT8 pSien0;  /* SIEN0 SCSI interrupt enable reg */
volatile UINT8 pSien1;  /* SIEN1 SCSI interrupt enable reg */
volatile UINT8 pSdid;   /* SID ID SCSI destination ID register */
volatile UINT8 pScnt1;  /* SCTNL1 SCSI control register 1 */
volatile UINT8 pScnt0;  /* SCTNL0 SCSI control register 0 */
volatile UINT8 pScnt2;  /* SCTNL2 SCSI control register 2 */
volatile UINT8 pScnt3;  /* SCTNL3 SCSI control register 3 */
volatile UINT8 pSreg;   /* General purpose reg */
volatile UINT8 pSsid;   /* SCSI selector ID */
volatile UINT8 pScntl1; /* SCTNL1 SCSI control register 1 */
volatile UINT8 pScntl0; /* SCTNL0 SCSI control register 0 */
volatile UINT8 pScntl2; /* SCTNL2 SCSI control register 2 */
volatile UINT8 pScntl3; /* SCTNL3 SCSI control register 3 */
volatile UINT pScratcha; /* General purpose scratch Pad a */
volatile UINT pScratchb; /* General purpose scratch Pad b */
volatile UINT pAdder;   /* Sum output of internal adder */
volatile UINT pSist0;  /* SCSI interrupt status 0 */
volatile UINT pSist1;  /* SCSI interrupt status 1 */
volatile UINT pSlpar;  /* SCSI longitudinal parity */
volatile UINT pMacnt1; /* SCSI Memory access control */
volatile UINT pGpCnt1; /* SCSI General purpose control */
volatile UINT pStime0; /* SCSI timer 0 */
volatile UINT pStime1; /* SCSI timer 1 */
volatile UINT pRespid; /* Response ID */
volatile UINT pRespid1; /* Response ID */
volatile UINT pSocl;   /* SOLL SCSI output control latch reg */
volatile UINT pSodl;   /* SODL SCSI output data latch reg */
volatile UINT pSxfer;  /* SODL SCSI transfer register */
volatile UINT pScid;   /* SCID SCSI chip ID register */
volatile UINT pSbcl;   /* SBCL SCSI bus control lines reg */
volatile UINT pSbdl;   /* SBDL SCSI bus data lines register */
volatile UINT pSidl;   /* SIDL SCSI input data latch reg */
volatile UINT pSfbr;   /* SFBR SCSI first byte received reg */
volatile UINT pSstat2; /* SSTAT2 SCSI status register 2 */
volatile UINT pSstat1; /* SSTAT1 SCSI status register 1 */
volatile UINT pSstat0; /* SSTAT0 SCSI status register 0 */
volatile UINT pSstat;  /* DSTAT DMA status register */
volatile UINT pBsa;    /* DSA data structure address */
volatile UINT pCtest3; /* CTEST3 chip test register 3 */
volatile UINT pCtest2; /* CTEST2 chip test register 2 */
volatile UINT pCtest1; /* CTEST1 chip test register 1 */
volatile UINT pCtest0; /* CTEST0 chip test register 0 */
volatile UINT pTemp;   /* TEMP temporary holding register */
volatile UINT pIstat;  /* ISTAT interrupt status register */
volatile UINT pDfifo;  /* DFIFO DMA FIFO control register */
volatile UINT pDcmd;   /* DBC SIOP command register 8bits */
volatile UINT pDbcc;   /* DCMD SIOP command reg (24Bits Reg) */

(continued on next page)
The first field in each xxx_SCSI_CTRL structure is the SCSI_CTRL structure, which is filled in by the SCSI device driver initialization routines, xxxCtrlCreate() and xxxCtrlInit(), during system startup. This structure provides the interface between the SCSI device driver and scsiLib. The information in the SCSI_CTRL structure includes the procedure handles with which scsiLib routines access the required set of driver routines. Example 8–3 shows the definition of SCSI_CTRL, from yourCloneTree/h/scsiLib.h.
Example 8–3  SCSI Driver Data Structure SCSI_CTRL

typedef struct scsiCtrl /* SCSI_CTRL - generic SCSI controller info */
{
    SEMAPHORE ctrlMutexSem; /* semaphore for exclusive access */
    SEMAPHORE ctrlSyncSem; /* semaphore for waiting on I/O interrupt */
    VOIDFUNCPTR scsiBusReset; /* function for resetting the SCSI bus */
    FUNCPTR scsiTransact; /* function for managing a SCSI transaction */
    FUNCPTR scsiDevSelect; /* function for selecting a SCSI device */
    FUNCPTR scsiBytesIn; /* function for SCSI input */
    FUNCPTR scsiBytesOut; /* function for SCSI output */
    FUNCPTR scsiDmaBytesIn; /* function for SCSI DMA input */
    FUNCPTR scsiDmaBytesOut; /* function for SCSI DMA output */
    FUNCPTR scsiBusPhaseGet; /* function returning the current bus phase */
    VoidFuncPtr scsiSyncMsgConvert; /* function for converting extended sync */
    VOIDFUNCPTR scsiSetAtn; /* function to set atn */
    UINT clkPeriod; /* period of the controller clock (nsec) */
    int scsiCtrlBusId; /* SCSI bus ID of this SCSI controller */
    int scsiPriority; /* priority of task when doing SCSI I/O */
    int scsiBusPhase; /* current phase of SCSI */
    typedef struct
    {
        INT disconnect; /* globally enable / disable disconnect */
        INT infoOnDevArr[MAX_SCSI_PHYS_DEVS]; /* info on dev. attached to this controller */
        int physDevArr[MAX_SCSI_PHYS_DEVS]; /* physical device info */
        FuncPtr scsiSyncMsgConvert; /* function for converting extended sync */
        VOIDFUNCPTR scsiSetAtn; /* function to set atn */
    } SCSI_PHYS_DEV;
} SCSI_CTRL;

The xxx_SCSI_CTRL controller structure in turn contains an array of SCSI_PHYS_DEV structures, one for each physical device on the SCSI bus, which are initialized when the required call to the SCSI library routine scsiPhysDeviceCreate() is made, from usrScsiConfig() in usrConfig.c, for each device on the SCSI bus. The SCSI_PHYS_DEV structure represents physical device information. Example 8–4 shows the definition of SCSI_PHYS_DEV, from yourCloneTree/h/scsiLib.h.
Example 8–4  SCSI Driver Data Structure SCSI_PHYS_DEV

/* NOTE: Not all of the fields in the following structure */
/* are currently supported or used. */

typedef struct /* SCSI_PHYS_DEV - SCSI physical device info */
{
    SEMAPHORE devMutexSem; /* semaphore for exclusive access */
    SEMAPHORE devSyncSem; /* semaphore for waiting on I/O interrupt */
    int scsiDevBusId; /* device’s address on SCSI bus */
    int scsiDevLUN; /* device’s logical unit number */
    UINT selTimeOut; /* device select time-out (units var.) */
    UINT8 scsiDevType; /* SCSI device type */
    TBOOL disconnect; /* whether device supports disconnect */
    TBOOL expectDisconnect; /* TRUE if a disconnect is expected */
    TBOOL xmitsParity; /* TRUE if dev. xmits parity */
    TBOOL useIdentify; /* whether to use identify message at select */
    TBOOL syncXfer; /* TRUE for synchronous data transfer */
    int syncXferOffset; /* synchronous xfer maximum offset */
    int syncXferPeriod; /* synchronous xfer mininum period */
    UINT8 *savedTransAddress; /* saved data pointer address */
    int savedTransLength; /* saved data pointer length */
    BOOL resetFlag; /* set TRUE when dev reset sensed */
    FUNCPTR postResetRtn; /* routine to call when dev has been reset */

    char devVendorID [VENDOR_ID_LENGTH + 1]; /* vendor ID in ASCII */
    char devProductID [PRODUCT_ID_LENGTH + 1]; /* product ID in ASCII */
    char devRevLevel [REV_LEVEL_LENGTH + 1]; /* revision level in ASCII */
}

(continued on next page)
Example 8–4 (Cont.)  SCSI Driver Data Structure SCSI_PHYS_DEV

```c
BOOL removable; /* whether medium is removable */
UINT8 lastSenseKey; /* last sense key returned by dev */
UINT8 lastAddSenseCode; /* last additional sense code returned by dev */
int controlByte; /* vendor unique control byte for commands */
int numBlocks; /* number of blocks on the physical device */
int blockSize; /* size of an SCSI disk sector */
struct scsiCtrl *pScsiCtrl; /* ptr to dev's SCSI controller info */
SCSI_DEV_STATUS devStatus; /* status of most recent device operation */
/* list of outgoing messages */
UINT8 msgOutArray [MAX_MSG_OUT_BYTES + 1]; /* msg[0] reserved for identify */
int msgLength; /* number of bytes in outgoing message */
BOOL extendedSense; /* whether device returns extended sense */
UINT8 *pReqSenseData; /* ptr to last REQ SENSE data returned by dev */
int reqSenseDataLength; /* size of REQ SENSE data array */
/* ptr to first block dev created on device */
struct scsiBlkDevList blkDevList;
struct scsiBlkDev *pScsiBlkDev;
int useMsgout; /* whether to send a message out at select */
int syncReserved1; /* Reserved for hardware sync register value */
int syncReserved2; /* Reserved for hardware sync register value */
SCSI_TRANSACTION *pScsiXaction; /* current scsi action requested */
} SCSI_PHYS_DEV;
```

8.3 Driver Operations

Table 8–3 lists the routines common to most or all implementations of a VxWorks SCSI driver.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxBusPhaseGet()</td>
<td>Return the current SCSI bus phase</td>
</tr>
<tr>
<td>xxxBytesIn()</td>
<td>Input SCSI device data</td>
</tr>
<tr>
<td>xxxBytesOut()</td>
<td>Output SCSI device data</td>
</tr>
<tr>
<td>xxxCtrlCreate()</td>
<td>Create a control structure for SCSI controller xxx</td>
</tr>
<tr>
<td>xxxCtrlInit()</td>
<td>Initialize a control structure for SCSI controller xxx</td>
</tr>
<tr>
<td>xxxDevSelect()</td>
<td>Select a SCSI device</td>
</tr>
</tbody>
</table>

(continued on next page)
Table 8–3 (Cont.) SCSI Driver Required Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxDmaBytesIn()</td>
<td>Input SCSI device data using DMA (required only for DMA support)</td>
</tr>
<tr>
<td>xxxDmaBytesOut()</td>
<td>Output SCSI device data using DMA (required only for DMA support)</td>
</tr>
<tr>
<td>xxxIntr()</td>
<td>Handle an interrupt for SCSI controller xxx</td>
</tr>
<tr>
<td>xxxMsgInAck()</td>
<td>Acknowledge an incoming message</td>
</tr>
<tr>
<td>xxxScsiBusReset()</td>
<td>Reset the SCSI bus</td>
</tr>
<tr>
<td>xxxScsiTransact()</td>
<td>Execute a SCSI command (omitted in the normal case, in which scsiLib controls data transfer; required for interfacing directly with the controller)</td>
</tr>
<tr>
<td>xxxSelTimeOutCvt()</td>
<td>Convert a timeout value</td>
</tr>
<tr>
<td>xxxShow()</td>
<td>Display the values of readable SCSI device registers</td>
</tr>
</tbody>
</table>

xxxCtrlCreate() and xxxCtrlInit() are driver initialization routines; they are user callable, but typically are called from system initialization code during VxWorks system startup.

The xxxCtrlCreate() routine is called once, before using the SCSI device, to create the device’s XXX_SCSI_CTRL data structure. The xxxCtrlInit() routine is called after the XXX_SCSI_CTRL structure is created, while the device is inactive, to initialize or reinitialize the data structure. Typically it also resets all devices attached to the bus.

The SCSI driver create and initialize routines also fill in the driver’s SCSI_CTRL structure, described in Section 8.2, and make the driver accessible to the higher-level scsiLib module that implements most of the SCSI protocol.

Several implementations of these routines are documented in the VxWorks reference material:

- ncr5390CtrlCreate()  Fujitsu NCR 53C90 Advanced SCSI Controller (ASC)
- ncr5390CtrlInit()    
- ascShow()            
- ncr710CtrlCreate()   NCR 53C710 SCSI I/O Processor (SIOP)
- ncr710CtrlInit()     
- ncr710Show()         

Writing a SCSI Device Driver 8–15
ncr810CtrlCreate() NCR 53C810 SCSI I/O Processor (PSIOP)
ncr810CtrlInit() 
ncr810Show()
wd33c93CtrlCreate() Western Digital wd33c93 SCSI-Bus Interface Controller (SBIC)
wd33c93CtrlInit() 
wd33c93Show()

xxx BusPhaseGet(), xxxBytesIn(), xxxBytesOut(), xxxDevSelect(), xxx_dmaBytesIn(), xxx_dmaBytesOut(), xxx_MsgInAck(), xxx_ScsiBusReset(), xxx_ScsiTransact(), and xxx_SELTimeOutCvt() are local routines that are identified to, and accessed by, the scsiLib routine library. Their names are filled into the SCSI_CTRL data structure during driver initialization.

Note

Most SCSI device drivers, including the NCR 53C90 ASC driver ncr5390Lib, do not include the xxx_ScsiTransact routine, but instead set the scsiTransact procedure handle to point to the scsiTransact() routine in scsiLib. The scsiTransact() routine then uses other routines provided by the driver for controlling the data transfer:

- xxxDevSelect()
- xxxBytesIn()
- xxxBytesOut()
- xxx_dmaBytesIn()
- xxx_dmaBytesOut()
- xxxBusPhaseGet()
- xxx_MsgInAck()

However, some SCSI devices, for example NCR 53C810 PSIOP, control all phases of a SCSI transaction in their firmware. In such cases, the driver provides the xxx_ScsiTransact() routine to interface directly with the controller, rather than relying on the SCSI library. For example, ncr810Lib provides the ncr810_ScsiTransact() routine, which invokes a script to perform SCSI transactions. xxxDevSelect() and the other routines listed above may be omitted from such a driver and the corresponding SCSI_CTRL procedure handles set to NULL.

xxx_Intr() is an interrupt handler routine; it is accessed through an interrupt vector that is connected using the intConnect() routine during driver initialization.

xxxShow() is a routine for displaying device registers; it is user callable, but is provided mainly for debugging.
The following sections describe each routine and the typical actions it performs. These descriptions are generalized and inclusive, based on SCSI drivers currently provided by VxWorks; not all the actions described apply to all SCSI devices.

In this section, the notation “xxx” or “XXX” is used to represent the SCSI device's two-character mnemonic in routine and structure names.

---

### xxxBusPhaseGet()

Return the current SCSI bus phase

**C Syntax**

```c
#include <scsiLib.h>

LOCAL STATUS xxxBusPhaseGet (SCSI_CTRL *pScsiCtrl,
                  int timeOutInUsec,
                  int *pBusPhase);
```

**Arguments**

- **pScsiCtrl**
  Specifies a pointer to a structure containing SCSI controller information for the device.

- **timeOutInUsec**
  Specifies the number of microseconds to wait for the information before timing out; specify 0 to wait indefinitely.

- **pBusPhase**
  Specifies a location to receive the current SCSI bus phase value.

**Description**

This routine, referenced through the scsiBusPhaseGet function pointer, returns a value indicating the current phase of the SCSI bus.

This routine is not called directly by user code.
SCSI Driver Operations

Return Value

OK.

xxxBytesIn()

Input SCSI device data

C Syntax

```c
#include <scsiLib.h>
LOCAL STATUS xxxBytesIn (SCSI_PHYS_DEV *pScsiPhysDev,
    char *pBuffer,
    int bufLength,
    int scsiPhase);
```

Arguments

**pScsiPhysDev**

Specifies a pointer to a structure containing the SCSI physical device information for the device from which input data should be returned.

**pBuffer**

Specifies a pointer to the buffer to receive input data.

**bufLength**

Specifies the length of the input buffer, or maximum number of bytes to transfer.

**scsiPhase**

Specifies a value indicating the SCSI bus phase of the transfer; valid values are defined in `scsiLib.h`.

Description

This routine, referenced through the `scsiBytesIn` function pointer, inputs data from a specified SCSI device. The routine should determine if a driver-specific DMA routine, `xxxDmaBytesIn()`, is available by checking the `scsiDmaBytesIn` function pointer. Based on whether a function pointer or NULL is found, the routine passes control to either the DMA routine or to a programmed input routine.

This routine is not called directly by user code.
**SCSI Driver Operations**

**Return Values**

OK if successful.
ERROR if not successful.

---

**xxxBytesOut()**

Output SCSI device data

**C Syntax**

```c
#include <scsiLib.h>
LOCAL STATUS xxxBytesOut (SCSI_PHYS_DEV *pScsiPhysDev,
                   char *pBuffer,
                   int bufLength,
                   int scsiPhase);
```

**Arguments**

- `pScsiPhysDev`
  Specifies a pointer to a structure containing the SCSI physical device information for the device to which output data should be written.

- `pBuffer`
  Specifies a pointer to the buffer containing the output data.

- `bufLength`
  Specifies the length of the output buffer, or number of bytes to transfer.

- `scsiPhase`
  Specifies a value indicating the SCSI bus phase of the transfer; valid values are defined in `scsiLib.h`.

**Description**

This routine, referenced through the `scsiBytesOut` function pointer, outputs data to a specified SCSI device. The routine should determine if a driver-specific DMA routine, `xxxDmaBytesOut()`, is available by checking the `scsiDmaBytesOut` function pointer. Based on whether a function pointer or NULL is found, the routine passes control to either the DMA routine or to a programmed output routine.
SCSI Driver Operations

This routine is not called directly by user code.

Return Values

OK if successful.
ERROR if not successful.

XXXCtrlCreate()

Create a control structure for SCSI controller xxx

C Syntax

```c
#include <xxx.h>

XXX_SCSI_CTRL *xxxCtrlCreate (UINT8 *baseAdrs,
                                 int regOffset,
                                 UINT clkPeriod,
                                 FUNCPTR xxxDmaBytesIn,
                                 FUNCPTR xxxDmaBytesOut,
                                 struct adp *adpt);
```

Arguments

- **baseAdrs**
  Specifies the base address of the device — the address at which the CPU would access the lowest (BDID) device register.

- **regOffset**
  Specifies the address offset, in bytes, between consecutive registers. This must be a power of two; for example, 1, 2, 4 and so on.

- **clkPeriod**
  Specifies the period, in nanoseconds, of the signal-to-device CLK input; used only for select timeouts.

- **xxxDmaBytesIn**
  Specifies a board-specific function to handle SCSI DMA input; if NULL (0), program transfer mode is used. For some chips, DMA is possible only during SCSI data in/out phases. The interface to a DMA routine specified by xxxDmaBytesIn or xxxDmaBytesOut must be of the form:
SCSI Driver Operations

STATUS xxxDmaBytes{In, Out} (SCSI_PHYS_DEV *pScsiPhysDev,
UINT8 *pBuffer,
int bufLength);

where pScsiPhysDev specifies a pointer to the SCSI physical device
information, pBuffer specifies a pointer to the data buffer, and bufLength
specifies the number of bytes to transfer.

**xxxDmaBytesOut**

Specifies a board-specific function to handle SCSI DMA output; if NULL
(0), program transfer mode is used. For some chips, DMA is possible only
during SCSI data in/out phases. The interface to a DMA routine specified by
xxxDmaBytesOut must be of the form shown above under xxxDmaBytesIn.

**adpt**

Specifies the address of the adapter structure for this device. You can look up
the address of a device’s adapter structure by calling the adpLookup() routine,
as described in the VxWorks Programmer’s Guide.

Description

This routine creates the SCSI_CTRL data structure needed by all other
routines in the library. It should be called only once for a given SCSI controller,
and should be the first SCSI driver routine called.

After calling this routine, at least one call to xxxCtrlInit() should be made
before any SCSI transaction is initiated using the SCSI chip.

The routine arguments listed above are generalized and representative.
Parameters can be substituted, added, or deleted, according to the needs of the
driver. For example, the NCR 53C90 ASC driver additionally requires callers
to pass an interrupt vector number, a controller type, and an interrupt routine
address. The NCR 53C810 PSIO P driver requires callers to pass only three
arguments, one of which is the frequency of the controller clock.

This routine is global and should be documented in a target-independent reference page, yourCloneTree/man/man2/xxxCtrlCreate.2. You can use
an existing xxxCtrlCreate.2 manpage as a starting point and modify the
manpage as necessary to match the implementation in your SCSI driver. Also,
use an existing *Lib.1 manpage, such as ncr5390Lib.1 or ncr810Lib.1, to
create a library overview manpage, describing the general functionality of your
SCSI driver.
SCSI Driver Operations

Return Values

A pointer to the device control structure, if successful.
NULL if memory is unavailable or there are bad parameters.

xxxCtrlInit()

Initialize a control structure for SCSI controller xxx

C Syntax

```
#include <xxx.h>

STATUS xxxCtrlInit ( XXX_SCSI_CTRL *pXxx,
         int scsiCtrlBusId,
         UINT defaultSelTimeOut,
         int scsiPriority);
```

Arguments

```
pXxx
Specifies a pointer to the XXX_SCSI_CTRL information structure created with xxxCtrlCreate().

scsiCtrlBusId
Specifies the SCSI bus ID, 0 through 7, of the device. The ID value is somewhat arbitrary; typically, the value 7, or highest priority, is used.

defaultSelTimeOut
Specifies the default timeout, in microseconds, for selecting a SCSI device attached to this controller; called a default since the timeout can be specified for each device (see the manual entry for scsiPhysDevCreate()). The recommended value 0 specifies SCSI_DEF_SELECT_TIMEOUT (250 milliseconds). The maximum timeout possible is approximately 3 seconds. Values that exceed the maximum revert to the maximum.

scsiPriority
Specifies the priority, ranging from 0 through 255, to which a task is set when performing a SCSI I/O transaction. Specify -1 to indicate that the priority should not be altered during SCSI transactions.
```
SCSI Driver Operations

Description

This routine initializes the SCSI_CTRL control structure, after the structure is created with xxxCtrlCreate(). The control structure must be initialized before the device can be used. The xxxCtrlInit() routine can be called more than once if needed, but should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

This routine is global and should be documented in a target-independent reference page, yourCloneTree/man/man2/xxxCtrlInit.2. You can use an existing xxxCtrlInit.2 manpage as a starting point and modify the manpage as necessary to match the implementation in your SCSI driver.

Return Values

OK if successful.

ERROR if any parameters are out of range.

xxxDevSelect()

Select a SCSI device

C Syntax

#include <scsiLib.h>

LOCAL STATUS xxxDevSelect (SCSI_PHYS_DEV *pScsiPhysDev,
                          SCSI_TRANSACTION *pScsiXAction);

Arguments

pScsiPhysDev
Specifies a pointer to a structure containing the SCSI physical device information for the device to be selected.

pScsiXAction
Specifies a pointer to appropriate SCSI transaction information. The SCSI_TRANSACTION structure, used by xxxScsiDevSelect() and xxxScsiTransact() to pass transaction information, is defined in yourCloneTree/h/scsiLib.h as follows:
SCSI Driver Operations

/* structure to pass to ioctl call to execute an SCSI command*/
typedef struct /* SCSI_TRANSACTION - information about a SCSI transaction */
{
    UINT8 *cmdAddress; /* address of SCSI command bytes */
    int cmdLength; /* length of command in bytes */
    UINT8 *dataAddress; /* address of data buffer */
    int dataDirection; /* direction of data transfer */
    int dataLength; /* length of data buffer in bytes (0=no data) */
    int addLengthByte; /* if != -1, byte # of additional length */
    UINT8 statusByte; /* status byte returned from target */
    int cmdTimeout; /* number of usec for this command timeout */
} SCSI_TRANSACTION;

Description

This routine, referenced through the scsiDevSelect function pointer, selects a target SCSI device to execute a SCSI command. The device controls the phases of the SCSI bus until the command completes.

This routine is not called directly by user code.

Return Values

OK if successful.
ERROR if the specified device could not be selected.

xxxxDmaBytesIn()

Input SCSI device data using DMA (optional)

C Syntax

#include <scsiLib.h>

LOCAL STATUS xxxxDmaBytesIn (SCSI_PHYS_DEV *pScsiPhysDev,
    UINT8 *pBuffer,
    int bufLength);

Arguments

pScsiPhysDev
Specifies a pointer to a structure containing the SCSI physical device information for the device from which input data should be returned.
SCSI Driver Operations

\textit{pBuffer}
Specifies a pointer to the buffer to receive input data.

\textit{bufLength}
Specifies the length of the input buffer, or maximum number of bytes to transfer.

\textbf{Description}
This routine, referenced through the scsiDmaBytesIn function pointer, is provided only if DMA is available, and desired, for the SCSI device. If you provide this function, you must specify it in the \texttt{xxxCtrlCreate()} routine for the device. Otherwise, you specify NULL for the function pointer and a programmed input routine is used for the device.

This routine is not called directly by user code. For more information on how it is called, see the \texttt{xxxBytesIn()} routine description.

\textbf{Return Values}
OK if successful.
ERROR if not successful.

\texttt{xxxDmaBytesOut()}
Output SCSI device data using DMA (optional)

\textbf{C Syntax}
\begin{verbatim}
#include <scsiLib.h>
LOCAL STATUS xxxDmaBytesOut (SCSI_PHYS_DEV *pScsiPhysDev, 
    UINT8 *pBuffer, 
    int bufLength);
\end{verbatim}

\textbf{Arguments}
\texttt{pScsiPhysDev}
Specifies a pointer to a structure containing the SCSI physical device information for the device to which output data should be written.

\texttt{pBuffer}
Specifies a pointer to the buffer containing the output data.
SCSI Driver Operations

(bufLength)
Specifies the length of the output buffer, or the number of bytes to transfer.

Description
This routine, referenced through the scsiDmaBytesOut function pointer, is provided only if DMA is available, and desired, for the SCSI device. If you provide this function, you must specify it in the xxxCtrlCreate() routine for the device. Otherwise, you specify NULL for the function pointer and a programmed output routine is used for the device.

This routine is not called directly by user code. For more information on how it is called, see the xxxBytesOut() routine description.

Return Values
OK if successful.
ERROR if not successful.

xxxIntr()
Handle an interrupt for SCSI controller xxx

C Syntax
VOID xxxIntr (XXX_SCSI_CTRL *pXxx);

Argument
pXxx
Specifies a pointer to the XXX_SCSI_CTRL structure for the device.

Description
This routine, which is invoked upon the occurrence of a device interrupt, handles the interrupt in an appropriate fashion.

If the device is on the VMEbus and does not implement automatic acknowledgement of VMEbus interrupts, this routine must manually acknowledge the VMEbus interrupt with a call to the sysBusIntAck() routine.

This routine is accessed through an interrupt vector and should not be called by user code.
**xxxMsgInAck()**

Acknowledge an incoming message

**C Syntax**

```c
#include <scsiLib.h>
LOCAL STATUS xxxMsgInAck (SCSI_CTRL *pScsiCtrl,
                          BOOL expectDisconn);
```

**Arguments**

- **pScsiCtrl**
  Specifies a pointer to a structure containing SCSI controller information for this device.

- **expectDisconn**
  Specifies TRUE if a disconnect is expected, FALSE otherwise.

**Description**

This routine, referenced through the scsiMsgInAck function pointer, acknowledges an incoming message by issuing a NEG_ACK command. If the received message does not imply a pending disconnect, the synchronization semaphore (defined in the SCSI_CTRL structure) is relinquished. This allows subsequent SCSI bus phases to be detected.

This routine is not called directly by user code.

**Return Value**

OK.
SCSI Driver Operations

xxxScsiBusReset()

Reset the SCSI bus

C Syntax
#include <scsiLib.h>
LOCAL VOID xxxScsiBusReset ( XXX_SCSI_CTRL *pXxx);

Argument
pXxx
Specifies a pointer to the XXX_SCSI_CTRL structure for the device.

Description
This routine, referenced through the scsiBusReset function pointer, asserts the RST (reset) line on the SCSI bus. It should result in all connected devices being reset to an initial state.
This routine is not called directly by user code.

xxxScsiTransact()

Execute a SCSI command

C Syntax
#include <scsiLib.h>
LOCAL STATUS xxxScsiTransact (SCSI_PHYS_DEV *pScsiPhysDev,
SCSI_TRANSACTION *pScsiXaction);

Arguments
pScsiPhysDev
Specifies a pointer to a structure containing the SCSI physical device information.
pScsiXAction
Specifies a pointer to appropriate SCSI transaction information. The
SCSI_TRANSACTION structure, used by xxxScsiDevSelect() and
xxxScsiTransact() to pass transaction information, is defined in
yourCloneTree/h/scsiLib.h as follows:
/* structure to pass to ioctl call to execute an SCSI command*/
typedef struct /* SCSI_TRANSACTION - information about a SCSI transaction */
{
    UINT8 *cmdAddress; /* address of SCSI command bytes */
    int cmdLength; /* length of command in bytes */
    UINT8 *dataAddress; /* address of data buffer */
    int dataDirection; /* direction of data transfer */
    int dataLength; /* length of data buffer in bytes (0=no data) */
    int addLengthByte; /* if != -1, byte # of additional length */
    UINT8 statusByte; /* status byte returned from target */
    int cmdTimeout; /* number of usec for this command timeout */
} SCSI_TRANSACTION;

Description
This routine, referenced through the scsiTransact function pointer, executes a
requested command.

Most SCSI device drivers do not include this routine, but instead set the
scsiTransact procedure handle to point to the scsiTransact() routine in
scsiLib. The scsiTransact() routine then uses other routines provided by
the driver for controlling the data transfer — xxxDevSelect(), xxxBytesIn(),
xxxBytesOut(), xxxDmaBytesIn(), xxxDmaBytesOut(), xxxBusPhaseGet(),
and xxxMsgInAck().

However, some SCSI devices control all phases of a SCSI transaction in their
firmware. In such cases, the driver provides the xxxScsiTransact() routine to
interface directly with the controller, rather than relying on the SCSI library.
(xxxDevSelect() and the other routines listed above may be omitted from the
driver and the corresponding SCSI_CTRL procedure handles set to NULL.)

The NCR 53C90 ASC device driver, ncr5390Lib, is an example of a driver that
uses the normally required routines and does not provide xxxScsiTransact().
By contrast, the NCR 53C810 SIO device driver, ncr810Lib, provides
the ncr810ScsiTransact() routine, which invokes a script to perform SCSI
transactions.

This routine is not called directly by user code.
SCSI Driver Operations

Return Values

OK if successful.
ERROR if not successful.

xxxSelTimeOutCvt()

Convert a timeout value

C Syntax

```c
#include <scsiLib.h>

LOCAL VOID xxxSelTimeOutCvt (SCSI_CTRL *pScsiCtrl,
                               UINT timeoutInUsec,
                               UINT *pTimeoutSetting);
```

Arguments

- **pScsiCtrl**
  Specifies a pointer to a structure containing SCSI controller information for the device.

- **timeoutInUsec**
  Specifies a timeout value, in microseconds, to be converted to units expected by the device.

- **pTimeoutSetting**
  Specifies the location to receive the converted timeout value.

Description

This routine, referenced through the scsiSelTimeOutCvt function pointer, converts a timeout value from microseconds to units expected by the device. If the device does not use a timeout value, for example to support a programmed timeout, specify the function pointer as NULL and omit this routine from the driver.

This routine is not called directly by user code.
**xxxShow()**

Display the values of readable SCSI device registers

**C Syntax**

```c
#include <xxx.h>

VOID xxxShow (XXX_SCSI_CTRL *pXxx);
```

**Argument**

`pXxx`

Specifies a pointer to the XXX_SCSI_CTRL structure created with `xxxCtrlCreate()`.

**Description**

This routine displays the state of the device registers in a user-friendly way. It is used primarily for debugging.

This routine is global and should be documented in a target-independent reference page, yourCloneTree/man/man2/xxxShow.2. You can use an existing `xxxShow.2` manpage as a starting point and modify the manpage as necessary to match the implementation in your SCSI driver.

**Note**

For some SCSI devices, this routine should be used with caution. For example, the `ncr810Lib` routine `ncr810Show()` should not be called during the execution of a SCSI script, as only the Istat register is safely readable at that time. The manpage you create for your `xxxShow()` routine should caution users as appropriate.
SCSI Driver Support Macros

8.4 Driver Support Macros

Table 8–4 lists macros that are provided by SCSI library header files and used to debug SCSI drivers.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_DEBUG_MSG()</td>
<td>Output a message from SCSI driver code</td>
</tr>
<tr>
<td>SCSI_INT_DEBUG_MSG()</td>
<td>Output a message from SCSI driver code</td>
</tr>
</tbody>
</table>

C Syntax

```c
#include <scsiLib.h>
SCSI_DEBUG_MSG (char *pBuffer);
```

C Syntax

```c
#include <scsiLib.h>
SCSI_INT_DEBUG_MSG (char *pBuffer);
```

Argument

`pBuffer`

Specifies the text to be output, typically supplied in a quoted string.

Description

These macros, defined in yourCloneTree/h/scsiLib.h, help with the debugging of SCSI device drivers by allowing you to output useful information from within the code being debugged.

To allow the use of these macros, the driver should IMPORT two global Boolean variables from `scsiLib`, `scsiDebug` and `scsiIntsDebug`. The variables can be set or reset from the VxWorks shell, but should not be enabled (set to TRUE) at the same time. Enabling these variables also causes more information about errors to be generated by the SCSI system.
SCSI Driver Support Macros

Examples

The examples below show the use of SCSI_DEBUG_MSG to report a driver error, and the use of SCSI_INT_DEBUG_MSG to output a vector value:

SCSI_DEBUG_MSG ("DevSelect error status\n");
SCSI_INT_DEBUG_MSG ("Vector = x\n", *pXxx->pVectorReg);
8.5 Sample SCSI Device Drivers

This section provides listings of two SCSI device drivers:

- The NCR 53C90 ASC SCSI driver, Section 8.5.1
- The NCR 53C810 PSIO P SCSI driver, along with an associated transaction scripts module, Section 8.5.2

The NCR 53C90 driver represents the traditional model of a SCSI device driver. It provides the normally required set of driver routines, which are called as needed by a SCSI library routine, scsiTransact(), to control a data transfer.

The NCR 53C810 driver represents an alternative type of SCSI device driver, in which the SCSI device controls all phases of a SCSI transaction in its firmware and the driver routine xxxScsiTransact() interfaces directly with the controller. This type of driver may provide an associated transaction script.

Note

The ncr810Lib driver is used on several Alpha platforms, including EB, AXPvme, and AXPpci platforms. The specific Alpha platform that uses the ncr5390Lib driver is not currently supported by VxWorks. However, this driver provides a valuable point of reference (and potential template) for supporting traditional SCSI controllers on Alpha platforms.

Both the ncr5390Lib and ncr810Lib drivers use adpLib I/O bus routines that make driver code more readily portable across Alpha targets; the majority of modifications to make either driver work on a new platform should be in BSP sysScsi routines, not in the driver itself.

If you have purchased a VxWorks Board Porting Kit or Source Kit, you can access, on line, the SCSI driver source modules in yourCloneTree/src/drv/scsi.

8.5.1 NCR 53C90 ASC SCSI Driver Listing

Example 8-5 provides a listing of the NCR 53C90 ASC SCSI driver, ncr5390Lib.c.
Example 8–5  NCR 53C90 ASC SCSI Driver Listing

/* ncr5390Lib.c - library for NCR 5390 Advanced SCSI Controller (ASC) */
/*
   Copyright (c) 1992
   DIGITAL Equipment Corporation, Maynard, Mass.
*/
/* Copyright 1989,1990 Wind River Systems, Inc. */
extern char copyright_wind_river[]; static char *copyright=copyright_wind_river;

/* DESCRIPTION
This is the I/O driver for the Fujitsu ncr5390 SCSI Protocol Controller (ASC).
It is designed to work in conjunction with scsiLib.

USER CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O
system. Two routines, however, must be called directly: ncr5390CtrlCreate(2)
to create a controller structure, and ncr5390CtrlInit(2) to initialize it.

INCLUDE FILES
ncr5390.h

SEE ALSO
"I/O System"
*/

#include "vxWorks.h"
#include "drv/scsi/ncr5390.h"
#include "memLib.h"
#include "iv.h"
#include "adpLib.h"
/* defines */
#define ASC_MAX_XFER_LENGTH ((UINT) (0xffff)) /* max data xfer length */
typedef NCR_5390_SCSI_CTRL ASC;
/* globals */
IMPORT BOOLEAN scsiDebug;
IMPORT BOOLEAN scsiIntsDebug;
/* forward declarations */

(continued on next page)
Example 8–5 (Cont.)  NCR 53C90 ASC SCSI Driver Listing

LOCAL STATUS ascBusIdGet (ASC *, int *);
LOCAL STATUS ascDevSelect ();
LOCAL STATUS ascBytesIn ();
LOCAL STATUS ascBytesOut ();
LOCAL STATUS ascMsgInAck ();
LOCAL STATUS ascIoctl ();
LOCAL STATUS ascBusPhaseGet ();
LOCAL VOID ascSelTimeOutCvt ();
LOCAL VOID ascScsiBusReset ();
struct adp *adap;
VOID ascCommand ();
VOID aschWInit ();

/*******************************************************************************
*  *
* ncr5390CtrlCreate - create a control structure for an ncr5390 ASC *
*  *
* Before using an ASC chip, a data structure must be created by calling this *
* routine. This routine should be called once and only once for a given ASC, *
* and should be the first routine called, since it alloc’s a structure needed *
* by all other routines in the library. *
*  *
* At least one call to ncr5390CtrlInit(2) should be made before any SCSI *
* transactions are initiated through the ASC, subsequent to the calling of *
* this routine. *
*  *
* A detailed description of the input parameters follows:
*  *
* .iP <baseAdrs>
* the address at which the CPU would access the lowest (BDID) *
* register of the ASC.
*  *
* .iP <regOffset>
* The address offset (bytes) to access consecutive registers. *
* (This must be a power of 2, for example, 1, 2, 4, etc.)
*  *
* .iP <clkPeriod>
* period in nanoseconds of the signal to ASC CLK input (only *
* used for select timeouts).
*  *
* .iP "<ascDmaBytesIn> and <ascDmaBytesOut>"
* board-specific routines to handle DMA input and output;
* if these are NULL (0), ASC program transfer mode is used. *
* DMA is possible only during SCSI data in/out phases. *
* The interface to these DMA routines must be of the form: *
* .CS *
* STATUS xxDmaBytes{In, Out} (pScsiPhysDev, pBuffer, bufLength)

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

*  
* SCSI_PHYS_DEV *pScsiPhysDev; * ptr to phys dev info *  
* UINT8 *pBuffer; * ptr to the data buffer *  
* int bufLength; * number of bytes to xfer *  
* .CE  
* Details on the expected functionality of such routines  
* will be documented elsewhere. *  
*  
* RETURNS: Pointer to ASC struct,  
* or NULL if memory unavailable or bad parameters. *  
*/

NCR_5390_SCSI_CTRL *ncr5390CtrlCreate (baseAdrs, regOffset, clkPeriod,  
ascDmaBytesIn, ascDmaBytesOut, ivec,type,  
intRoutine,adpt)  
{  
FAST ASC *pAsc; /* ptr to ASC info */  
/* verify parameters */  
adap = adpt; /* save adapter address globally for use by all functs */  
if (regOffset == 0)  
return ((ASC *) NULL);  
/* calloc the controller info structure; return ERROR if unable */  
pAsc = (ASC *) calloc (1, sizeof (ASC));  
if (pAsc == (ASC *) NULL)  
return ((ASC *) NULL);  
/* set chip type (90, 94) */  
pAsc->devType = type;  
/* connect up the interrupt */  
(void) intConnect (ivec, intRoutine, pAsc);  
/* fill in generic SCSI info for this controller */  
scsiCtrlInit (&pAsc->scsiCtrl);  

(continued on next page)
/* fill in ASC specific data for this controller */

pAsc->pTclReg = baseAdrs;
pAsc->pTchReg = baseAdrs + (0x1 * regOffset);
pAsc->pFifoReg = baseAdrs + (0x2 * regOffset);
pAsc->pCmdReg = baseAdrs + (0x3 * regOffset);
pAsc->pStatReg = baseAdrs + (0x4 * regOffset);
pAsc->pIntrReg = baseAdrs + (0x5 * regOffset);
pAsc->pStepReg = baseAdrs + (0x6 * regOffset);
pAsc->pFlgsReg = baseAdrs + (0x7 * regOffset);
pAsc->pCfg1Reg = baseAdrs + (0x8 * regOffset);
pAsc->pClkReg = baseAdrs + (0x9 * regOffset);
pAsc->pTestReg = baseAdrs + (0xa * regOffset);
pAsc->pCfg2Reg = baseAdrs + (0xb * regOffset);

if (pAsc->devType == ASC_NCR5394)
    pAsc->pCfg3Reg = baseAdrs + (0xc * regOffset);

pAsc->scsiCtrl.clkPeriod = clkPeriod;
pAsc->scsiCtrl.maxBytesPerXfer = ASC_MAX_XFER_LENGTH;
pAsc->scsiCtrl.scsiBusReset = (VOIDFUNCPTR) ascScsiBusReset;

pAsc->scsiCtrl.scsiTransact = (FUNCPTR) scsiTransact;
pAsc->scsiCtrl.scsiDevSelect = (FUNCPTR) ascDevSelect;
pAsc->scsiCtrl.scsiBytesIn = (FUNCPTR) ascBytesIn;
pAsc->scsiCtrl.scsiBytesOut = (FUNCPTR) ascBytesOut;
pAsc->scsiCtrl.scsiDmaBytesIn = (FUNCPTR) ascDmaBytesIn;
pAsc->scsiCtrl.scsiDmaBytesOut = (FUNCPTR) ascDmaBytesOut;
pAsc->scsiCtrl.scsiBusPhaseGet = (FUNCPTR) ascBusPhaseGet;
pAsc->scsiCtrl.scsiMsgInAck = (FUNCPTR) ascMsgInAck;
pAsc->scsiCtrl.scsiSelTimeOutCvt = (VOIDFUNCPTR) ascSelTimeOutCvt;

pAsc->pDevToSelect = (SCSI_PHYS_DEV *) NULL;

return (pAsc);

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

/*******************************************************************************
* ncr5390CtrlInit - initialize a control structure for an ncr5390 ASC
* After an ASC structure is created with ncr5390CtrlCreate(2), but
* before using the ASC, 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, since the specified
* configuration is written to the ASC.
* Before returning, this routine pulses RST (reset) on the SCSI bus, thus
* resetting all attached devices.
* A detailed description of the input parameters follows:
* .iP <pAsc>
* pointer to the NCR_5390_SCSI_CTRL structure created with
* ncr5390CtrlCreate.
* .iP <scsiCtrlBusId>
* the SCSI bus ID of the ASC; somewhat arbitrary, the value 7,
* or highest priority, is conventional. Must be in range 0 - 7.
* .iP <defaultSelTimeOut>
* the timeout (in microseconds) for selecting a SCSI device
* attached to this controller; called default since the
* timeout may be specified per device (see
* scsiPhysDevCreate(2)). The recommended value 0
* specifies SCSI_DEF_SELECT_TIMEOUT (250 millisec).
* The maximum timeout possible is approximately 3 seconds.
* Values over the maximum specify the maximum. See also Fujitsu
* documentation.
* .iP <scsiPriority>
* the priority to which a task is set when performing a SCSI
* transaction. Legal priorities are 0 to 255. Alternately,
* the value -1 indicates that the priority should not be
* altered during SCSI transactions.
* RETURNS: OK, or ERROR if out-of-range parameter(s).
* */

STATUS ncr5390CtrlInit (pAsc, scsiCtrlBusId, defaultSelTimeOut, scsiPriority)
    FAST NCR_5390_SCSI_CTRL *pAsc; /* ptr to ASC struct */
    FAST int scsiCtrlBusId;  /* SCSI bus ID of this ASC */
    FAST UINT defaultSelTimeOut; /* default device select time-out (usec) */
    int scsiPriority;        /* priority of a task when doing SCSI I/O */

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
UINT tempSelTimeOut; /* temp. value of select time-out (no units) */
FAST UINT tempClkPeriod; /* local copy of clock period */
int tempClkConvFactor; /* local copy of clock conversion factor */

/* verify scsiCtrlBusId and enter legal value in ASC structure */
if (scsiCtrlBusId < SCSI_MIN_BUS_ID || scsiCtrlBusId > SCSI_MAX_BUS_ID)
    return (ERROR);
else
    pAsc->scsiCtrl.scsiCtrlBusId = (UINT8) scsiCtrlBusId;

/* verify scsiPriority and enter legal value in ASC structure */
if (scsiPriority < NONE || scsiPriority > 0xff)
    return (ERROR);
else
    pAsc->scsiCtrl.scsiPriority = scsiPriority;

/* issue NO-OP (required after hardware reset) */
ascCommand (pAsc, NCR5390_CHIP_RESET);
taskDelay (1); /* was (1) */
if (pAsc->devType == ASC_NCR5394)
    ascCommand (pAsc, NCR5390_NOP);
else
    ascCommand (pAsc, NCR5390_DMA_OP | NCR5390_NOP);

/* set clock conversion factor */
tempClkPeriod = pAsc->scsiCtrl.clkPeriod;
if (tempClkPeriod >= 100) /* clock freq <= 10 Mhz */
    tempClkConvFactor = 2;
else if (tempClkPeriod >= 67) /* clock freq <= 15 Mhz */
    tempClkConvFactor = 3;
else if (tempClkPeriod >= 50) /* clock freq <= 20 Mhz */
    tempClkConvFactor = 4;
else /* clock freq > 20 Mhz */
    tempClkConvFactor = 5;

pAsc->clkCvtFactor = (UINT8) tempClkConvFactor;
adpWrite(adap,pAsc->pClkReg,0,1, tempClkConvFactor);

/* verify defaultSelTimeOut, convert it from usec to ASC register values, */
/* and enter value in ASC structure */
ascSelTimeOutCvt (&pAsc->scsiCtrl, defaultSelTimeOut, &tempSelTimeOut);
```

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
pAsc->defaultSelTimeOut = (UINT8) tempSelTimeOut;
/* disconnect not supported for now */
ascHwInit (pAsc); /* initialize the ASC hardware */
return (OK);
}

/*******************************************************************************
* ascDevSelect - attempt to select a SCSI physical device
* This routine is intended to be called from scsiLib, not directly.
* RETURNS: OK if device was successfully selected, otherwise ERROR.
* ARGUSED1
*/
LOCAL STATUS ascDevSelect (pScsiPhysDev, pScsiXaction)
FAST SCSI_PHYS_DEV *pScsiPhysDev; /* ptr to SCSI physical device info */
SCSI_TRANSACTION *pScsiXaction; /* ptr to SCSI transaction info */
{
FAST ASC *pAsc; /* ptr to ASC info */
int ascBusId; /* SCSI bus ID of the ASC */
UINT8 identMsg; /* for construction of the IDENTIFY message */
STATUS status; /* placeholder for status */
int sisciPhase; /* SCSI bus phase following the select */
int ix; /* loop index */
UINT8 *pCmdByte; /* ptr to a command byte */
UINT8 ascOpcode; /* ASC opcode byte */
UINT8 temp;
pAsc = (ASC *) pScsiPhysDev->pScsiCtrl;
if (!(ascBusIdGet (pAsc, &ascBusId) == ERROR) ||
    (ascBusId == pScsiPhysDev->scsiDevBusId))
{
    return (ERROR);
}
pAsc->pDevToSelect = pScsiPhysDev;
adpWrite(adap,pAsc->pBidReg,0,1,(UINT8) pScsiPhysDev->scsiDevBusId);
pScsiPhysDev->devStatus = SELECT_REQUESTED;
if (pScsiPhysDev->useIdentify) /* send an identify message */
{  
(continued on next page)
```

Writing a SCSI Device Driver 8–41
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```
identMsg = SCSI_MSG_IDENTIFY | 
(pAsc->scsiCtrl.disconnect ? SCSI_MSG_IDENT_DISCONNECT : 0) | 
(UINT8) pScsiPhysDev->scsiDevLUN;
adpWrite(adap,pAsc->pFifoReg,0,1,identMsg);
}
pCmdByte = pScsiXaction->cmdAddress;
for (ix = 0; ix < pScsiXaction->cmdLength; ix++){
    temp = *pCmdByte++;
adpWrite(adap,pAsc->pFifoReg,0,1,temp);
}
if (pScsiPhysDev->useIdentify)
    { /* XXX correct param ? */
        ascOpcode = (UINT8) NCR5390_ATN_SELECT;
        else
            ascOpcode = (UINT8) NCR5390_STOP_SELECT;
    }else
        ascOpcode = (UINT8) NCR5390_SELECT;
if (pAsc->devType == ASC_NCR5394)
    ascOpcode = (UINT8) NCR5390_SELECT;
ascCommand (pAsc, ascOpcode);
if (pAsc->devType == ASC_NCR5394){
    ascCommand (pAsc, (UINT8) NCR5390_SELECTION_ENBL);
    ascOpcode = (UINT8) NCR5390_SELECT;
}
semTake (&pAsc->scsiCtrl.ctrlSyncSem, WAIT_FOREVER);
```
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

if (!(pAsc->savedIntrReg == NCR5390_DISCONNECTED))
{
    if (!(pAsc->savedIntrReg == (NCR5390_BUS_SERVICE | NCR5390_FUNC_COMPLETE)))
    {
        printErr (*ascDevSelect: Unknown chip state, pAsc = %x\n", pAsc);
        printErr (*ascdevselect: ascOpcode = %x\n", ascOpcode);
        printErr (*statReg = 0x%02x, stepReg = 0x%02x, intrReg = 0x%02x\n", pAsc->savedStatReg, pAsc->savedStepReg, pAsc->savedIntrReg);
        ascCommand (pAsc, NCR5390_FIFO_FLUSH);
        return (ERROR);
    }
    else
    {
        switch ((int) ascOpcode)
        {
        case NCR5390_SELECT:
            if (pAsc->savedStepReg == 4)
            {
                return (OK);
            }
            else
            {
                if (pAsc->savedStepReg == 2)
                    errnoSet (S_scsiLib_PHASE_CHANGE_TIMEOUT);
                else if (pAsc->savedStepReg == 3)
                    errnoSet (S_scsiLib_EARLY_PHASE_CHANGE);
                break;
            }
        case NCR5390_ATN_SELECT:
            if (pAsc->savedStepReg == 4)
            {
                return (OK);
            }
            else
            {
                if ((pAsc->savedStepReg == 0) || (pAsc->savedStepReg == 2))
                    errnoSet (S_scsiLib_PHASE_CHANGE_TIMEOUT);
                else if (pAsc->savedStepReg == 3)
                    errnoSet (S_scsiLib_EARLY_PHASE_CHANGE);
                break;
            }
        case NCR5390_STOP_SELECT:
            if (pAsc->savedStepReg == 1)
            {
                return (OK);
            }
            else
            {
                (continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

if (pAsc->savedStepReg == 0)
    errnoSet (S_scsiLib_PHASE_CHANGE_TIMEOUT);
break;
}
ascCommand (pAsc, NCR5390_FIFO_FLUSH);
return (ERROR);
}
else
{
    ascCommand (pAsc, NCR5390_FIFO_FLUSH);
    errnoSet (S_scsiLib_SELECT_TIMEOUT);
    return (ERROR);
}

/*******************************************************************************
 * ascAtnSet - assert the ATN line to request a MESSAGE OUT phase on SCSI
 */
LOCAL VOID ascAtnSet (pScsiCtrl)
    FAST SCSI_CTRL *pScsiCtrl; /* ptr to ASC struct */
{
    ascCommand ((ASC *) pScsiCtrl, NCR5390_SET_ATTENTION);
}
/*******************************************************************************
 * ascMsgInAck - de-assert the ACK line to accept message
 * This routine should be called when an incoming message has been read and
 * accepted. If the incoming message did not imply an impending disconnect,
 * give the synchronization semaphore, so that subsequent phase can be
 * detected.
 * RETURNS: OK.
 */
LOCAL STATUS ascMsgInAck (pScsiCtrl, expectDisconn)
    FAST SCSI_CTRL *pScsiCtrl; /* ptr to ASC info */
    BOOL expectDisconn; /* whether a disconnect is expected */
{
    ascCommand ((ASC *) pScsiCtrl, NCR5390_MSG_ACCEPTED);
    semTake (&pScsiCtrl->ctrlSyncSem, WAIT_FOREVER);
}

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

    return (OK);
}

/*******************************************************************************
* ascProgBytesOut - output bytes to SCSI using ‘program’ transfer
* RETURNS: OK if successful, otherwise ERROR.
* INTERNAL
* Error handling is inadequate.
*/
STATUS ascProgBytesOut (pScsiPhysDev, pBuffer, bufLength, scsiPhase)
    SCSI_PHYS_DEV *pScsiPhysDev; /* ptr to physical device info */
    FAST UINT8 *pBuffer; /* ptr to the data buffer */
    FAST int bufLength; /* number of bytes to be transferred */
    int scsiPhase; /* phase of the transfer */
{
    FAST ASC *pAsc; /* ptr to ASC info */
    FAST int ix; /* loop index */
    BOOL noOpIssued; /* set TRUE after first DMA NO-OP command */
    unsigned char tempx; /* hold values from reading registers */
    SCSI_DEBUG_MSG ("ascProgBytesOut: scsiPhase = %s, bufLength = %d\n",
        scsiPhaseNameGet (scsiPhase), bufLength);
    pAsc = (ASC *) pScsiPhysDev->pScsiCtrl;
    noOpIssued = FALSE;
    adpRead(adap,pAsc->pFlgsReg,0,1,&tempx);
    if (tempx & NCR5390_MORE_DATA)
        SCSI_DEBUG_MSG ("ascProgBytesOut: FIFO Not Empty.\n");
    sysWbFlush();
    while (bufLength >= NCR5390_FIFO_DEPTH)
    {
        for (ix = 0; ix < NCR5390_FIFO_DEPTH; ix++)
        {
            tempx = *pBuffer++;
            adpWrite(adap,pAsc->pFifoReg,0,1,tempx);
        }
    }
    (continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
if (!noOpIssued)
{
    ascCommand (pAsc, NCR5390_DMA_OP | NCR5390_NOP);
    noOpIssued = TRUE;
}

ascCommand (pAsc, NCR5390_INFO_TRANSFER);

if (semTake (&pAsc->scsiCtrl.ctrlSyncSem, sysClkRateGet ()) == ERROR)
    return (ERROR);

/* XXX check error conditions */
bufLength -= NCR5390_FIFO_DEPTH;

if (bufLength > 0)
{
    while (bufLength > 0)
    {
        tempx = *pBuffer++;
        adpWrite(adap,pAsc->pFifoReg,0,1,tempx);
        bufLength--;
    }

    if (!noOpIssued)
    {
        ascCommand (pAsc, NCR5390_DMA_OP | NCR5390_NOP);
        noOpIssued = TRUE;
    }

    ascCommand (pAsc, NCR5390_INFO_TRANSFER);

    if (semTake (&pAsc->scsiCtrl.ctrlSyncSem, sysClkRateGet ()) == ERROR)
        return (ERROR);

    /* XXX check error conditions */
}
return (OK);
```

/* *****************************************************
* ascProgBytesIn - input bytes from SCSI using 'program' transfer
* RETURNS: OK if successful, otherwise ERROR.
* INTERNAL
* Error handling is inadequate.
*/

(continued on next page)
Example 8–5 (Cont.)  NCR 53C90 ASC SCSI Driver Listing

```c
STATUS ascProgBytesIn (pScsiPhysDev, pBuffer, bufLength, scsiPhase)
SCSI_PHYS_DEV *pScsiPhysDev; /* ptr to physical device info */
FAST UINT8 *pBuffer; /* ptr to the data buffer */
FAST int bufLength; /* number of bytes to be transferred */
int scsiPhase; /* phase of the transfer */
{
    FAST ASC *pAsc; /* ptr to ASC info */
    unsigned char tempx;
    SCSI_DEBUG_MSG ("ascProgBytesIn: scsiPhase = %s, bufLength = %d\n",
                        scsiPhaseNameGet (scsiPhase), bufLength);

    pAsc = (ASC *) pScsiPhysDev->pScsiCtrl;
    adpRead(adap,pAsc->pFlgsReg,0,1,&tempx);
    if (tempx & NCR5390_MORE_DATA)
        SCSI_DEBUG_MSG ("ascProgBytesIn: FIFO Not Empty.\n");
    ascCommand (pAsc, NCR5390_DMA_OP | NCR5390_NOP);
    while (bufLength > 0)
    {
        ascCommand (pAsc, NCR5390_INFO_TRANSFER);
        if (semTake (&pAsc->scsiCtrl.ctrlSyncSem, sysClkRateGet ()) == ERROR)
        {
            SCSI_DEBUG_MSG("semtake failed in ascprogbytesin\n");
            return (ERROR);
        }
        /* XXX check error conditions */
        adpRead(adap,pAsc->pFifoReg,0,1,&tempx);
        *pBuffer++ = tempx;
        bufLength--;
    }
    SCSI_DEBUG_MSG("exiting ok from ascprogbytesin\n");
    return (OK);
}
```

*******************************************************************************
*
* ascBytesOut - branch to the appropriate output routine dependent on SCSI phase *
* RETURNS: OK if specified bytes were output successfully, otherwise ERROR.
*
(continued on next page)
LOCAL STATUS ascBytesOut (pScsiPhysDev, pBuffer, bufLength, scsiPhase)
FAST SCSI_PHYS_DEV *pScsiPhysDev; /* ptr to SCSI physical dev info */
FAST char *pBuffer; /* ptr to byte buffer for output */
int bufLength; /* number of bytes to be transferred */
int scsiPhase; /* SCSI bus phase of the transfer */
{
FAST ASC *pAsc; /* ptr to ASC info */
STATUS status; /* local status for iterative xfers */
int xferLength; /* local length for iterative xfers */
pAsc = (ASC *) pScsiPhysDev->pScsiCtrl;
do
{
xferLength = min (bufLength, ASC_MAX_XFER_LENGTH);
if (((scsiPhase != SCSI_DATA_OUT_PHASE) ||
    (pAsc->scsiCtrl.scsiDmaBytesOut == NULL))
    
status = ascProgBytesOut (pScsiPhysDev, (UINT8 *) pBuffer,
    xferLength, scsiPhase);
} else
{
    status = (pAsc->scsiCtrl.scsiDmaBytesOut)
    (pScsiPhysDev, (UINT8 *) pBuffer, xferLength);
}
/* if ERROR was returned, exit now */
if (status != OK)
    break;
else
    pBuffer += ASC_MAX_XFER_LENGTH;
} while ((bufLength -= ASC_MAX_XFER_LENGTH) > 0);
return (status);
}
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

LOCAL STATUS ascBytesIn (pScsiPhysDev, pBuffer, bufLength, scsiPhase)
FAST SCSI_PHYS_DEV *pScsiPhysDev;  /* ptr to SCSI physical dev info */
FAST char *pBuffer;                  /* ptr to byte buffer for output */
int bufLength;                       /* number of bytes to be transferred */
int scsiPhase;                       /* SCSI bus phase of the transfer */
{
    FAST ASC *pAsc;                  /* ptr to ASC info */
STATUS status;                     /* local status for iterative xfers */
int xferLength;                    /* local length for iterative xfers */
int bufLengthCopy;                 /* buflength copied here */

    pAsc = (ASC *) pScsiPhysDev->pScsiCtrl;
    do
    {
        xferLength = min (bufLength, ASC_MAX_XFER_LENGTH);
        if ((scsiPhase != SCSI_DATA_IN_PHASE) ||
            (pAsc->scsiCtrl.scsiDmaBytesIn == NULL))
            { status = ascProgBytesIn (pScsiPhysDev, (UINT8 *) pBuffer,
                                          xferLength, scsiPhase);
            }
        else
            {
                status = (pAsc->scsiCtrl.scsiDmaBytesIn)
                          (pScsiPhysDev, (UINT8 *) pBuffer, xferLength);
            }
        /* if ERROR was returned, exit now */
        if (status != OK)
            break;
        else
            pBuffer += ASC_MAX_XFER_LENGTH;
    } while ((bufLength -= ASC_MAX_XFER_LENGTH) > 0);
    return (status);
}

*******************************************************************************
* ascBusPhaseGet - return the current SCSI bus phase in *pBusPhase
* INTERNAL
* Should implement timeout and do an errnoSet.
*/

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
LOCAL STATUS ascBusPhaseGet (pScsiCtrl, timeOutInUsec, pBusPhase)
FAST SCSI_CTRL *pScsiCtrl; /* ptr to SCSI controller info */
int timeOutInUsec; /* timeout in usec (0 == infinity) */
int *pBusPhase; /* ptr to returned bus phase */
{
    FAST ASC *pAsc; /* ptr to ASC info */
    pAsc = (ASC *) pScsiCtrl;
    if (pAsc->savedIntrReg & NCR5390_BUS_SERVICE)
    {
        *pBusPhase = pAsc->savedStatReg & NCR5390_PHASE_MASK;
        return (OK);
    }
    else if (pAsc->savedIntrReg & NCR5390_DISCONNECTED)
    {
        *pBusPhase = SCSI_BUS_FREE_PHASE;
        return (OK);
    }
    else
    return (ERROR);
}

/*******************************************************************************
* ascSelTimeOutCvt - convert a select time-out in microseconds to its
* equivalent ASC setting
* The conversion formula is given on p. 15 of the 1989 NCR 53C90A / 53C90B
* manual. Note that 0 translates to the standard setting of 250 usec. Also,
* the ASC accepts up to a 8 bit time-out, so a maximum value of 0xff is
* returned in *pTimeOutSetting.
*/
LOCAL VOID ascSelTimeOutCvt (pScsiCtrl, timeOutInUsec, pTimeOutSetting)
FAST SCSI_CTRL *pScsiCtrl; /* ptr to SCSI controller info */
FAST UINT timeOutInUsec; /* timeout in microseconds */
FAST UINT *pTimeOutSetting; /* time-out equivalent setting */
{
    FAST ASC *pAsc = (ASC *) pScsiCtrl; /* ptr to ASC info */
    FAST UINT tempSelTimeOut; /* temp. select time-out setting */
    if (timeOutInUsec == 0)
        timeOutInUsec = SCSI_DEF_SELECT_TIMEOUT;
    tempSelTimeOut = ((timeOutInUsec * 1000) / 
        (8192 * (pAsc->clkCvtFactor) * pScsiCtrl->clkPeriod)) + 1;
}
```

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

if (tempSelTimeOut > 0xff)
    tempSelTimeOut = 0xff;
*pTimeOutSetting = tempSelTimeOut;
}

/*******************************************************************************
 *
 * ascBusIdGet - get the current SCSI bus ID of the ASC.
 *
 * Copies the bus ID to <pBusId>.
 *
 * RETURNS:
 * OK if Bus ID register holds a legal value, otherwise ERROR.
 */
LOCAL STATUS ascBusIdGet (pAsc, pBusId)
    FAST ASC *pAsc; /* ptr to ASC info */
    FAST int *pBusId; /* ptr to returned bus ID */
{
    unsigned char tempx;
    adpRead(adap,pAsc->pCfg1Reg,0,1,&tempx);
    *pBusId = (int) tempx & NCR5390_OWN_ID_MASK;
    sysWbFlush();
    return (OK);
}

/*******************************************************************************
 *
 * ascXferCountSet - load the ASC transfer counter with the specified count
 *
 * RETURNS:
 * OK if count is in range 0 - 0xffff, otherwise ERROR.
 */
STATUS ascXferCountSet (pAsc, count)
    FAST ASC *pAsc; /* ptr to ASC info */
    FAST int count; /* count value to load */
{
    unsigned char temp;
    if (count < 0 || count > ASC_MAX_XFER_LENGTH)
        return (ERROR);
    else
    {
(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

temp = (UINT8) ((count >> 8) & 0xff);
adpWrite(adap,pAsc->pTchReg,0,1, temp);
adpWrite(adap,pAsc->pTclReg,0,1, (UINT8) (count & 0xff));
    return (OK);
}

/*******************************************************************************
* ascXferCountGet - fetch the ASC transfer count
* RETURNS: The value of the transfer counter is returned in *pCount.
*/
VOID ascXferCountGet (pAsc, pCount)
    FAST ASC *pAsc; /* ptr to ASC info */
    FAST int *pCount; /* ptr to returned value */
{
    unsigned char tempCount = 0;
    unsigned char tempCount1 = 0;
adpRead(adap,pAsc->pTclReg,0,1,&tempCount);
adpRead(adap,pAsc->pTchReg,0,1,&tempCount1);
tempCount |= (UINT) tempCount1 << 8; /* new */
sysWbFlush();
*pCount = tempCount;
}

/*******************************************************************************
* ascCommand - write a command code to the ASC Command Register
*/
VOID ascCommand (pAsc, cmdCode)
    FAST ASC *pAsc; /* ptr to ASC info */
    FAST int cmdCode; /* new command code */
{
    unsigned char temp = (UINT8) cmdCode;
adpWrite(adap,pAsc->pCmdReg,0,1,temp);
sysWbFlush();
}
(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

/*******************************************************************************
* ascScsiBusReset - assert the RST line on the SCSI bus
* Issue a SCSI Bus Reset command to the NCR 5390. This should put all devices
* on the SCSI bus in an initial quiescent state. */
LOCAL VOID ascScsiBusReset (pAsc)
    FAST ASC *pAsc; /* ptr to ASC info */
    {
        unsigned char temp;
        ascCommand (pAsc, NCR5390_BUS_RESET);
        taskDelay (1);
        if (pAsc->devType == ASC_NCR5394){
            temp = NCR5390_NOP;
            adpWrite(adap,pAsc->pCmdReg,0,1,temp);
            adpRead(adap,pAsc->pStatReg,0,1,&pAsc->savedStatReg);
            adpRead(adap,pAsc->pStepReg,0,1,&pAsc->savedStepReg);
            pAsc->savedStepReg &= 0x7;
            adpRead(adap,pAsc->pIntrReg,0,1,&pAsc->savedIntrReg);
            taskDelay (500); /* Allow devs to settle */
        }
    }
/*******************************************************************************
* ascIntr - interrupt service routine for the ASC
* NOMANUAL */
VOID ascIntr (pAsc) 
    ASC *pAsc; /* ptr to ASC info */
    {
        FAST SCSI_PHYS_DEV *pDevToSelect;
        unsigned char savedStatReg;
        unsigned char savedIntrReg;
        unsigned char savedStepReg;
        extern sysAscIntrClear();
        extern sysAscIntrEnable();

        (continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
adpRead(adap,pAsc->pStatReg,0,1,&savedStatReg);
sysWbFlush();
if (! (savedStatReg & 0x80)) return; /* If not real interrupt, exit */
adpRead(adap,pAsc->pStepReg,0,1,&savedStepReg);
sysWbFlush();
savedStepReg &= 0x7;
adpRead(adap,pAsc->pIntrReg,0,1,&savedIntrReg);
sysWbFlush();
pAsc->savedStatReg = savedStatReg;
pAsc->savedStepReg = savedStepReg;
pAsc->savedIntrReg = savedIntrReg;
SCSI_INT_DEBUG_MSG ("*** ASC INTERRUPT: ");
SCSI_INT_DEBUG_MSG ("statReg = 0x%02x, stepReg = 0x%02x, intrReg = 0x%02x\n",
    savedStatReg, savedStepReg, savedIntrReg);
if ((savedIntrReg & NCR5390_SELECTED) ||
    (savedIntrReg & NCR5390_ATN_SELECTED))
    /* this is currently not supported */
    SCSI_INT_DEBUG_MSG ("Selected ");
}
if (savedIntrReg & NCR5390_RESELECTED)
    /* this is currently not supported */
    SCSI_INT_DEBUG_MSG ("Reselected ");
}
if (savedIntrReg & NCR5390_FUNC_COMPLETE)
    /* Function Complete */
    SCSI_INT_DEBUG_MSG ("Function Complete ");
    /* if a selection attempt has been made, report success */
    if (((pDevToSelect = pAsc->pDevToSelect) != (SCSI_PHYS_DEV *) NULL) &&
        (pDevToSelect->devStatus == SELECT_IN_PROGRESS))
    {
        pDevToSelect->devStatus = SELECT_SUCCESSFUL;
        pAsc->pDevToSelect = (SCSI_PHYS_DEV *) NULL;
    }
```

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

    if (savedIntrReg & NCR5390_BUS_SERVICE)  
        {  
            SCSI_INT_DEBUG_MSG ("Bus Service ");  
        }  
    
    if (savedIntrReg & NCR5390_DISCONNECTED)  
        {  
            SCSI_INT_DEBUG_MSG ("Disconnect ");  
            /* if a device has been selected, then a timeout has occurred */  
            if (((pDevToSelect = pAsc->pDevToSelect) != (SCSI_PHYS_DEV *) NULL) &&  
                (pDevToSelect->devStatus == SELECT_IN_PROGRESS))  
                {  
                    pDevToSelect->devStatus = SELECT_TIMEOUT;  
                    pAsc->pDevToSelect = (SCSI_PHYS_DEV *) NULL;  
                }  
            else  
                {  
                    pAsc->scsiCtrl.scsiBusPhase = SCSI_BUS_FREE_PHASE;  
                }  
        }  
    
    if (savedIntrReg & NCR5390_ILLEGAL_CMD)  
        {  
            SCSI_INT_DEBUG_MSG ("Illegal Command (PANIC) ");  
        }  
    
    if (savedIntrReg & NCR5390 SCSI_RESET)  
        {  
            SCSI_INT_DEBUG_MSG ("Reset Detected ");  
        }  

    SCSI_INT_DEBUG_MSG ("
");  
    semGive (&pAsc->scsiCtrl.ctrlSyncSem);  
}

(continued on next page)
Example 8–5 (Cont.)  NCR 53C90 ASC SCSI Driver Listing
/* ********************** */
* ascHwInit - initialize the ASC chip to a known state
* This routine puts the ASC into a known quiescent state and issues a reset
* to the SCSI Bus if any signals are active, thus putting target devices in
* some presumably known state. Currently the initial state is not configurable
* and does not enable reselection.
* INTERNAL
* Needs to handle parity enable
*/
VOID ascHwInit (pAsc)
ASC *pAsc; /* ptr to an ASC info structure */
{
UINT8 tempCfg1Reg = (UINT8) 0;
unsigned char temp; /* build local copy of desired Configuration-1 Register */
temp = NCR5390_CHIP_RESET;
adpWrite(adap,pAsc->pCmdReg,0,1,temp); /* reset */
temp = NCR5390_NOP;
adpWrite(adap,pAsc->pCmdReg,0,1,temp); /* nop required */
tempCfg1Reg = ((UINT8) pAsc->scsiCtrl.scsiCtrlBusId) |
(pAsc->parityCheckEnbl ? NCR5390_PAR_CHECK_ENBL : 0) |
(pAsc->parityTestMode ? NCR5390_PAR_TEST_ENBL : 0) |
(pAsc->resetReportDsbl ? NCR5390_RESET_REP_DSBL : 0) |
(pAsc->slowCableMode ? NCR5390_SLOW_CABLE : 0);
/* write local copy of desired Configuration-1 Register to chip */
adpWrite(adap,pAsc->pCfg1Reg,0,1,tempCfg1Reg);

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
if (pAsc->devType == ASC_NCR5394) {
    temp = 8;
    adpWrite(adap, pAsc->pCfg2Reg, 0, 1, temp); /* Select SCSI 2 mode */
    temp = 0;
    adpWrite(adap, pAsc->pCfg3Reg, 0, 1, temp); /* 0 = IO ASIC */
    temp = 5;
    adpWrite(adap, pAsc->pClkReg, 0, 1, temp); /* Based on internal clock for SCSI */
    temp = 500;
    adpWrite(adap, pAsc->pTmoReg, 0, 1, temp);
    temp = 0;
    adpWrite(adap, pAsc->pFlgsReg, 0, 1, temp); /* Sync off */
    temp = NCR5390_FIFO_FLUSH;
    adpWrite(adap, pAsc->pCmdReg, 0, 1, temp); /* Flush Fifo */
    temp = NCR5390_SELECTION_ENBL;
    adpWrite(adap, pAsc->pCmdReg, 0, 1, temp); /* ready for selection */
}
if (pAsc->scsiCtrl.disconnect)
    /* XXX set it up */;
ascScsiBusReset (pAsc);
}
```

 /******************************************************************************
  * ascShow - Display values of all readable ncr5390 (ASC) registers
  * Displays state of the ASC registers in a user-friendly way. Primarily for use
  * during debugging.
  */
VOID ascShow (pAsc)
    FAST ASC *pAsc; /* ptr to ASC info */
{
    int busId;
    FAST UINT8 tempReg;
    int xferCount;
    unsigned char temp;

    (continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```c
adpRead(adap,pAsc->pTclReg,0,1,&temp);
printf("pTclReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pTchReg,0,1,&temp);
printf("pTchReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pFifoReg,0,1,&temp);
printf("pFifoReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pCmdReg,0,1,&temp);
printf("pCmdReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pStatReg,0,1,&temp);
printf("pStatReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pIntrReg,0,1,&temp);
printf("pIntrReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pStepReg,0,1,&temp);
printf("pStepReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pFlgsReg,0,1,&temp);
printf("pFlgsReg = 0x%02x\n", temp); sysWbFlush();
adpRead(adap,pAsc->pCfg1Reg,0,1,&temp);
printf("pCfg1Reg = 0x%02x\n", temp); sysWbFlush();
```

```c
#if FALSE
  /* Display current SCSI Bus ID */
  
  if (ascBusIdGet (pAsc, &busId) == OK)
    printf("SCSI Bus ID: %-5d\n", busId);
  else
    printf("SCSI Bus ID: ERROR\n");
  /* Display ASC Control Register */
  
adpRead(adap,pAsc->pSctlReg,0,1,&tempReg);
sysWbFlush();
  printf("SCTL (0x%02x): ", (int) tempReg);
  printf("%s", (tempReg & ASC_SCTL_RESET_AND_DSBL) ? "ascDisabl " : "");
  printf("%s", (tempReg & ASC_SCTL_CTRL_RESET) ? "ctrlReset " : "");
  printf("%s", (tempReg & ASC_SCTL_DIAG_MODE) ? "diagMode " : "");
  printf("%s", (tempReg & ASC_SCTL_ARBIT_ENBL) ? "arbitEnbl " : "");
  printf("%s", (tempReg & ASC_SCTL_PARITY_ENBL) ? "paritEnbl " : "");
  printf("%s", (tempReg & ASC_SCTL_SELECT_ENBL) ? "selecEnbl " : "");
  printf("%s", (tempReg & ASC_SCTL_RESELECT_ENBL) ? "reselEnbl " : "");
  printf("%s", (tempReg & ASC_SCTL_INT_ENBL) ? "intsEnbl " : "");
  printf("\n");
```

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

/* Display Command Register */
adpRead(adap,pAsc->pScmdReg,0,1,&tempReg);
printf("SCMD (0x%02x): ", (int) tempReg);
switch (tempReg & ASC_SCMD_CMD_MASK)
{
      case ASC_SCMD_BUS_RELEASE: printf("busRelease "); break;
      case ASC_SCMD_SELECT: printf("select "); break;
      case ASC_SCMD_RESET_ATN: printf("resetAtn "); break;
      case ASC_SCMD_SET_ATN: printf("setAtn "); break;
      case ASC_SCMD_XFER: printf("transfer "); break;
      case ASC_SCMD_XFER_PAUSE: printf("xferPause "); break;
      case ASC_SCMD_RESET_ACK_REQ: printf("resAckReq "); break;
      case ASC_SCMD_SET_ACK_REQ: printf("setAckReq "); break;
}
printf("%s", (tempReg & ASC_SCMD_RESET_OUT) ? "resetOut ": "");
printf("%s", (tempReg & ASC_SCMD_INTERCEPT_XFER) ? "intcptXfr ": "");
printf("%s", (tempReg & ASC_SCMD_PRG_XFER) ? "progXfer ": "");
printf("%s", (tempReg & ASC_SCMD_TERM_MODE) ? "termMode ": "");
printf("\n");
/* Display Transfer Mode Register */
adpRead(adap,pAsc->pTmodReg,0,1,&tempReg);
printf("TMOD (0x%02x): ", (int) tempReg);
printf("%s", (tempReg & ASC_TMOD_SYNC_XFER) ? "syncMode ": "");
printf("asyncMode ");
/* JCC Should display offset and period params also */
printf("\n");
/* Display Interrupt Sense Register */
adpRead(adap,pAsc->pIntrReg,0,1,&tempReg);
printf("INTS (0x%02x): ", (int) tempReg);
printf("%s", (tempReg & ASC_INTS_SELECTED) ? "selected ": "");
printf("%s", (tempReg & ASC_INTS_RESELECTED) ? "reselect ": "");
printf("%s", (tempReg & ASC_INTS_DISCONNECT) ? "disconnct ": "");
printf("%s", (tempReg & ASC_INTS_COM_COMPLETE) ? "comCompI ": "");
printf("%s", (tempReg & ASC_INTS_SERVICE_REQ) ? "servReq ": "");
printf("%s", (tempReg & ASC_INTS_TIMEOUT) ? "timeout ": "");
printf("%s", (tempReg & ASC_INTS_HARD_ERROR) ? "hardError ": "");
printf("%s", (tempReg & ASC_INTS_RESET_COND) ? "resetCond ": "");
printf("\n");

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

/* Display Phase Sense Register */
tempReg = *pAsc->pPsnsReg;
printf ("PSNS (0x%02x): ", (int) tempReg);
printf ("%s", (tempReg & ASC_PSNS_REQ) ? "REQ1 " : "req0 ");
printf ("%s", (tempReg & ASC_PSNS_ACK) ? "ACK1 " : "ack0 ");
printf ("%s", (tempReg & ASC_PSNS_ATN) ? "ATN1 " : "atn0 ");
printf ("%s", (tempReg & ASC_PSNS_SEL) ? "SEL1 " : "sel0 ");
printf ("%s", (tempReg & ASC_PSNS_BS) ? "BSY1 " : "bsy0 ");
printf ("%s", (tempReg & ASC_PSNS_MS) ? "MSG1 " : "msg0 ");
printf ("%s", (tempReg & ASC_PSNS_C_D) ? "C_D1 " : "c_d0 ");
printf ("%s", (tempReg & ASC_PSNS_I_O) ? "I_O1 " : "i_o0 ");
printf ("\n");

/* Display ASC Status Register */
adpRead(adap,pAsc->pSstsReg,0,1,&tempReg);
printf ("SSTS (0x%02x): ", (int) tempReg);
switch (tempReg & ASC_SSTS_OPER_STAT_MASK)
{
    case ASC_SSTS_NO_CONNECT_IDLE: printf ("noConIdle "); break;
    case ASC_SSTS_SELECT_WAIT: printf ("waitSelec "); break;
    case ASC_SSTS_TARGET_MANUAL: printf ("trgManual "); break;
    case ASC_SSTS_RESELECT_EXEC: printf ("reseleExec "); break;
    case ASC_SSTS_TARGET_XFER: printf ("targXfer "); break;
    case ASC_SSTS_INITIATOR_Manual: printf ("iniManual "); break;
    case ASC_SSTS_INITIATOR_WAIT: printf ("iniWait "); break;
    case ASC_SSTS_SELECT_EXEC: printf ("selecExec "); break;
    case ASC_SSTS_INITIATOR_XFER: printf ("iniXfer "); break;
    default: printf ("operUndef "); break;
}
printf ("%s", (tempReg & ASC_SSTS_SCSI_RESET) ? "scsiReset ": ");
printf ("%s", (tempReg & ASC_SSTS_TC_0) ? "xferCnt=0 ": ");
switch (tempReg & ASC_SSTS_DREG_STAT_MASK)
{
    case ASC_SSTS_DREG_EMPTY: printf ("dregEmpty "); break;
    case ASC_SSTS_DREG_PARTIAL: printf ("dregPartl "); break;
    case ASC_SSTS_DREG_FULL: printf ("dregFull "); break;
    default: printf ("dregUndef "); break;
}
printf ("\n");

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

/* Display ASC Error Status Register */
adpRead(adap, pAsc->pSerrReg, 0, 1, &tempReg);
printf("SERR (0x%02x): ", (int) tempReg);
switch (tempReg & ASC_SERR_PAR_ERROR_MASK)
{
    case ASC_SERR_NO_PAR_ERROR: printf("noParErr "); break;
    case ASC_SERR_PAR_ERROR_OUT: printf("outParErr "); break;
    case ASC_SERR_PAR_ERROR_IN: printf("inpParErr "); break;
    default: printf("pErrUndef "); break;
}
printf("%s", (tempReg & ASC_SERR_TC_PARITY) ? "tcParErr ": ");
printf("%s", (tempReg & ASC_SERR_PHASE_ERROR) ? "phaseErr ": ");
printf("%s", (tempReg & ASC_SERR_SHORT_PERIOD) ? "shortPeri ": ");
printf("%s", (tempReg & ASC_SERR_OFFSET_ERROR) ? "offsetErr ": ");
printf("\n");
/* Display Phase Control Register */
adpRead(adap, pAsc->pPctlReg, 0, 1, &tempReg);
printf("PCTL (0x%02x): ", (int) tempReg);
printf("%s", (tempReg & ASC_PCTL_BF_INT_ENBL) ? "bfIntEnbl ": "bfIntDsbl ");
switch (tempReg & ASC_PCTL_PHASE_MASK)
{
    case ASC_PCTL_DATA_OUT: printf("phDataOut "); break;
    case ASC_PCTL_DATA_IN: printf("phDataIn "); break;
    case ASC_PCTL_COMMAND: printf("phCommand "); break;
    case ASC_PCTL_STATUS: printf("phStatus "); break;
    case ASC_PCTL_UNUSED_0: printf("phUnused0 "); break;
    case ASC_PCTL_UNUSED_1: printf("phUnused1 "); break;
    case ASC_PCTL_MESS_OUT: printf("phMessOut "); break;
    case ASC_PCTL_MESS_IN: printf("phMessIn "); break;
}
printf("\n");
/* Display Modified Byte Counter */
adpRead(adap, pAsc->pMbcReg, 0, 1, &tempReg);
printf("MBC (0x%02x): ", (int) tempReg);
printf("%-10d", (int) tempReg);
printf("\n");
/* Display Transfer Counter */
ascXferCountGet(pAsc, &xferCount);

(continued on next page)
Example 8–5 (Cont.) NCR 53C90 ASC SCSI Driver Listing

```
sysWbFlush();
printf("XFER COUNT : ");
printf("0x%06x =%10d", xferCount, xferCount);
printf("\n");
#endif /* FALSE */
```

8.5.2 NCR 53C810 PSIOP SCSI Driver Listing

Example 8–6 provides a listing of the NCR 53C810 PSIOP SCSI driver, ncr810Lib.c. Example 8–7 lists an associated transaction scripts module, ncr810init.c.

Example 8–6 NCR 53C810 PSIOP SCSI Driver Listing

```
/* ncr810Lib.c - NCR 53C810 SCSI I/O Processor (PSIOP) library */

/* Copyright 1989-1992 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
DESCRIPTION
This is the I/O driver for the NCR 53C810 SCSI I/O Processor (SIOP).
It is designed to work in conjunction with scsiLib. This driver runs in
conjunction with a script program for the NCR 53C810 chip.
The script uses the NCR 53C810 DMA function for data transfers.
This driver supports cache functions through cacheLib.

USER CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O
system. Three routines, however, must be called directly: ncr810CtrlCreate()
to create a controller structure, and ncr810CtrlInit() to initialize it.
The NCR 53C810 hardware registers need to be configured according to
the hardware implementation. If the default configuration is not proper
the routine ncr810SetHwRegister() should be used to properly configure
the registers.

(continued on next page)```
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

INTERNAL
This driver supports the connect/disconnect capability. But this feature is invalidated in the driver code in ncr810CtrlInit() :
.cs
/
disconnect is supported for now, but this feature is not included
@ in this revision.
@
psIop->scsiCtrl.disconnect = FALSE; /@ TRUE to support disconnect @/;
.cs

to validate the disconnect, this field must be set to TRUE (and recompile)
and the field <pScsiPhysDev->useIdentify> must be also set to TRUE.
It also support the synchronous message. The new scsiSyncTarget() routine
in the scsiLib could be use to set up a synchronous protocol between the target
and the initiator.
There are debug variables to trace events in the driver.
<scsiDebug> scsiLib debug variable, trace event in scsiLib, ncr810ScsiPhase(),
and ncr810ScsiTransact().
<scsiIntsDebug> prints interrupt informations.
<ncr810LogErr> print out error case and dump the ncr810 register contents.
<ncr810Debug> Print out informations during the SCSI transaction.

INCLUDE FILES
ncr810.h, ncr810Script.h
SEE ALSO: scsiLib
.pG "I/O System"
*/
#include "vxWorks.h"
#include "drv/scsi/ncr810.h"
#include "memLib.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "logLib.h"
#include "semLib.h"
#include "ioLib.h"
#include "errnoLib.h"
#include "cacheLib.h"
#include "drv/timer/timerDev.h"
/* defines */
/* The following two macros are used for accessing the I/O registers on the
ncr810 chip. Any routine using these must have the variable pSiop defined
and pointing to the PSIOp structure for the particular 810 device. */

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
#define NCR810_READ(o,n,d) adpRead (pSiop->adapter,\
    pSiop->baseIO,\
    (o),\
    (n),\
    (d))

#define NCR810_WRITE(o,n,v) adpWrite(pSiop->adapter,\
    pSiop->baseIO,\
    (o),\
    (n),\
    (v));\
    sysWbFlush();

typedef NCR_810_SCSI_CTRL SIOP;

#define SIOP_MAX_XFER_LENGTH ((UINT) (0x00ffffff)) /* max data xfer length */

/* Time out used in the semtake in the driver. Because this is a global value
 * that not a good way to handle a time out for a scsi command. This should
 * be done in the scsiLib for each command running and processed by the
 * driver during the devSync semTake(). */
#define DEF_TIMEOUT 0x1000000

/* ncr810 debug level */
#define NCR810_MSG
    printErr

#define NCR810_DEBUG_MSG
    if (ncr810Debug)
        NCR810_MSG

#define LOG_ERR
    if (ncr810LogErr == 1)
        printErr

#define LOG_ERR_CTX
    if (ncr810LogErr == 1) ncr810Show((SCSI_CTRL *)pSiop)

/* ncr810 debug phase mismatch level */
#define NCR810_MM_MSG
    logMsg

#define NCR810_MISMATCH_MSG
    if (ncr810MmatchLog == 1)
        NCR810_MM_MSG

BOOL ncr810ScriptStarted = FALSE;
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* External */
IMPORT BOOL scsiDebug;
IMPORT BOOL scsiIntsDebug;
IMPORT UINT * relocation; /* relocation table information */
IMPORT UINT * selWithAtn_idx; /* index for the first entry point */
IMPORT UINT * waitSelect; /* Idle script entry point */
IMPORT UINT * selWithAtn; /* First scsi Script routine atn set */
IMPORT UINT * selWithoutAtn; /* First scsi Script routine ,no atn */
IMPORT UINT * ackMsg1; /* retry entry Script routine for extended msg */
IMPORT UINT * ackMsg2; /* retry entry Script routine msg after data */
IMPORT UINT * checkPhData; /* Entry Script routine to ack a message */
IMPORT UINT * endAbort; /* Entry Script when a msg out as reset */
IMPORT UINT * asortPh; /* Entry script to sort phase before data */
IMPORT UINT * checkNewCmd; /* Entry point to start a new cmd at intr lvl */
IMPORT UINT * contTransact;
IMPORT UINT *script_ptr[]; /* table of all entry in the script */
IMPORT UINT sysScsiConvertAddr(unsigned int *);/* convert long to 32 bit addr */
IMPORT UINT sysScsiConvertAddr(ulong int *);/* convert long to 32 bit addr */

/* sysLib.c must contain a function to convert addresses to 32 bits for use
by the ncr810 scripts processor which can access only 32 bit addresses.
If this driver is being used on a 32 bit processor, then the routine can
probably just return as output, the same value passed as input. For 64
bit processors, the routine must convert the 64 bit address to a 32 bit
physical address (an appropriate PCI address). */

/* This part allows logging some information at interrupt level. It’s a
* conditional compile feature. This is useful to track disconnect/reconnect
* operation at interrupt level.
* A show routine, ncr810LogShow() print and increment the <pLogMemShow>
* pointer.
*/
#if FALSE /* FALSE to disable interrupt log memory trace
* at the compile time.
*/
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* To enable interrupt log memory capability at the compile time. Allocated
memory with a malloc (size set by LOG_SIZE_MEM) and assign this return
value to <pNcr810Deb>, and pLogMemCount, set ncr810LogMem to 1, and
ncr810LogCount to 0. */
#endif /* TRUE/FALSE */
#endif NCR810_INT_LOG_MEM /* Interrupt trace enabled */
#define LOG_SIZE_MEM 0x10000 /* Memory log size */
int ncr810LogMem = 0; /* enable/disable memory log at the run time */
UINT *pNcr810Deb; /* pointer to the beginning of log memory */
(UINT *pLogMemShow; /* internal pointer for incremental show of */
/* log memory */
int ncr810LogCount = 0; /* memory log fifo counter */
/* macro to write memory log element */
#define LOG_MEO(o,n) \
NCR810_READ(o,n,*{pNcr810Deb + ncr810LogCount});\ 
ncr810LogCount++;\ 
if (ncr810LogCount >= (LOG_SIZE_MEM /4))\   
ncr810LogCount = 0;
#define LOG_MEMA(s) \
*(pNcr810Deb + ncr810LogCount)= s; \ 
ncr810LogCount++;\ 
if (ncr810LogCount >= (LOG_SIZE_MEM /4))\   
ncr810LogCount = 0;

(continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOC SCSI Driver Listing

#define LOG_MEM_INTR\  
  if (ncr810LogMem == 1) \  
  { \  
    LOG_MEMA((UINT)ncr810CountIntr) \  
    LOG_MEMA((UINT)ncr810LogCount) \  
    LOG_MEM(pSiop->pDsp) \  
    LOG_MEMA((UINT)pSiop->pNcrCtl) \  
    LOG_MEMA((UINT)pSiop->pNcrCtlCmd) \  
    LOG_MEMA((UINT)pSscsiPhysDev) \  
    LOG_MEMA((UINT)pSiop->pNcrCtl->pSscsiPhysDev) \  
    LOG_MEMA((UINT)pSiop->pNcrCtl->pDataLong) \  
    LOG_MEMA(pSiop->pNcrCtl->dBtct) \  
    LOG_MEMA((UINT)pSiop->pNcrCtl->dBtct) \  
    LOG_MEMA((UINT)pSiop->pNcrCtl->dBtct) \  
    LOG_MEMA((UINT)(0x99999999)) \  
  } \  
#endif /* NCR810_INT_LOG_MEM */

/* variables */
TBOOL siopSSStep; /* Global state of single step script */
/* execution */
int ncr810CountIntr = 0; /* Global interrupt counter */
/* Debug variables use by the debug macros */
int ncr810Debug = 0; /* <1> enable debug msg */
int ncr810MmatchLog = 0; /* <1> enable mismatch screen log (int lvl) */
int ncr810LogErr = 0; /* <1> enable print task level screen error */
  * log (task lvl). */
/* default timeout value use in semTake (devSyncSem.. */
int ncr810TimeOut = DEF_TIMEOUT;
/* forward declarations */

(continued on next page)
VOID ncr810StepEnable (SIOP *pSiop, TBOOL);
LOCAL void ncr810SelTimeOutCvt (SCSI_CTRL *pScsiCtrl, UINT timeOutInUseC, 
    UINT *pTimeOutSetting);
LOCAL STATUS ncr810ScsiTransact(SCSI_PHYS_DEV *pScsiPhysDev, 
    SCSI_TRANSACTION *pScsiXaction);
STATUS ncr810ScsiPhase(SCSI_PHYS_DEV *pScsiPhysDev, 
    SCSI_TRANSACTION *pScsiXaction);
STATUS ncr810SetHwRegister (SIOP *pSiop, NCR810_HW_REGS *pHwRegs);
void ncr810HwInit (SIOP *pSiop);
STATUS ncr810SyncMsgConvert (SCSI_PHYS_DEV *pScsiPhysDev, UINT8 *pMsgIn, 
    int *pSyncVal, int *pOffsetVal);
LOCAL UINT *ncr810RejectRecv (SIOP *pSiop, SCSI_PHYS_DEV *pScsiPhysDev, 
    NCR_CTL *pNcrCtl);
UINT *ncr810ExtMsgRecv (SIOP *pSiop, SCSI_PHYS_DEV *pScsiPhysDev, 
    NCR_CTL *pNcrCtl, UINT8 *msgIn);
LOCAL UINT *ncr810PhaseMismatch (SIOP *pSiop, SCSI_PHYS_DEV *pScsiPhysDev, 
    NCR_CTL *pNcrCtl, UINT phaseRequested);
LOCAL STATUS ncr810BuildMsgOut (SCSI_PHYS_DEV *pScsiPhysDev);
STATUS ncr810FirstStartPhase (SIOP *pSiop, NCR_CTL *pNcrCtl, 
    SCSI_PHYS_DEV *pScsiPhysDev, int timeOut);
LOCAL STATUS ncr810StartPhase (SIOP *pSiop, NCR_CTL *pNcrCtl, 
    UINT *scriptAddress, int timeOut);
void ncr810StartScript (SIOP *pSiop, NCR_CTL *pNcrCtl, 
    SCSI_PHYS_DEV *pScsiPhysDev, UINT scriptEntry);
LOCAL STATUS ncr810CheckStatRegs (SIOP *pSiop, SCSI_PHYS_DEV *pScsiPhysDev, 
    UINT **localStart);
LOCAL void ncr810FlushCache (SIOP *pSiop, NCR_CTL *pNcrCtl);

Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIO SCSI Driver Listing

/***************************************************************
* ncr810CtrlCreate - create a control structure for the SIOP
*
* This routine creates an SIOP data structure and must be called before
* using an SIOP chip. It should be called only once for a given SIOP.
* Since it allocates memory for a structure needed by all routines in
* ncr810lib, it must be called before any other routines in the library.
* After calling this routine, at least one call to ncr810CtrlInit()
* should be performed before any SCSI transactions are initiated using
* the SIOP.
*
* A detailed description of the input parameters follows:
* 
* .iP <baseAdrs>
* the address at which the CPU would access the lowest (BDID)
* register of the SIOP.
* 
* .iP <freqValue>
* the value at the SIOP SCSI CLK input. This is used to determine
* the clock period for the scsi core of the chip and the synchronous
* divider value for synchronous transfer.
* It is important to have the right timing on the SCSI bus.
* <freqValue> is defined as the SCSI clock input value in nanoseconds
* multiply by 100.
* Several are defined in ncr810.h as follows:
* .CS
* NCR810_1667MHZ 5998 /* 16.67Mhz chip */
* NCR810_20MHZ 5000 /* 20Mhz chip */
* NCR810_25MHZ 4000 /* 25Mhz chip */
* NCR810_3750MHZ 2666 /* 37.50Mhz chip */
* NCR810_40MHZ 2500 /* 40Mhz chip */
* NCR810_50MHZ 2000 /* 50Mhz chip */
* NCR810_66MHZ 1515 /* 66Mhz chip */
* NCR810_6666MHZ 1500 /* 66.66Mhz chip */
* .CE
* 
* RETURNS: A pointer to NCR_810_SCSI_CTRL structure, or NULL if memory
* is unavailable or there are bad parameters.
*/

NCR_810_SCSI_CTRL *ncr810CtrlCreate
{
    UINT8 *baseAdrs, /* base address of the SIOP */
    UINT freqValue, /* clock controller period (nsec*100) */
    struct adp *adpt /* adapter structure pointer */
}

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

\{
  FAST SIOP *pSiop;  /* ptr to SIOP info */
  FAST NCR_CTL_CTXT *pNcrCtxt;  /* ptr to SIOP context */

  /* verify parameters */
  if (baseAdrs == ((UINT8 *) NULL))
    return ((SIOP *) NULL);
  if ((freqValue < 0) ||
      (freqValue > NCR810_1667MHz))
    return ((SIOP *) NULL);

  /* check that dma buffers are write coherent */
  if (!CACHE_DMA_IS_WRITE_COHERENT ())
    return ((SIOP *) NULL);

  /* calloc the controller info structure; return ERROR if unable */
  pSiop = (SIOP *) cacheDmaMalloc (sizeof (SIOP));
  if (pSiop == (SIOP *) NULL)
    return ((SIOP *) NULL);

  bzero ( (char *)pSiop,sizeof (SIOP));
  pSiop->adapter = adpt;  /* set ptr to adapter str for use by all routines */
  pSiop->baseIO = baseAdrs;  /* set the base address of the PSIOP */

  /* cacheDmaMalloc the data share info structure; return ERROR if unable */
  pNcrCtxt = (NCR_CTL_CTXT *) cacheDmaMalloc (sizeof(NCR_CTL_CTXT));
  if (pNcrCtxt == (NCR_CTL_CTXT *) NULL)
  {printErr("ncr810CtrlCreate :malloc NCR_CTL_CTX context failed
    (void) cacheDmaFree ((char *) pSiop);
    return((SIOP *) NULL);
  }

  /* fill in generic SCSI info for this controller */
  scsiCtrlInit (&pSiop->scsiCtrl);

  /* Init NcrCtxt pointer in siop structure */
  pSiop->pNcrCtxt = pNcrCtxt;

  /* fill in SIOP specific data for this controller */
  /* Fill in the register offsets - the adpWrite and adpRead routines will use the offsets to derive the correct address */
\}

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
pSiop->pSien0 = OFF_SIEN0;
pSiop->pSien0Int = OFF_SIEN0;
pSiop->pSien1 = OFF_SIEN1;
pSiop->pSdid = OFF_SDID;
pSiop->pScntl1 = OFF_SCNTL1;
pSiop->pScntl10 = OFF_SCNTL0;
pSiop->pScntl12 = OFF_SCNTL2;
pSiop->pScntl13 = OFF_SCNTL3;
pSiop->pScocl = OFF_SOCL;
pSiop->pSodl = OFF_SODL;
pSiop->pSxfer = OFF_SXFER;
pSiop->pSicl = OFF_SCID;
pSiop->pSbl = OFF_SBCL;
pSiop->pSbd1 = OFF_SBDL;
pSiop->pSid1 = OFF_SIDL;
pSiop->pSfb = OFF_SFB;
pSiop->pSstat2 = OFF_SSTAT2;
pSiop->pSstat1 = OFF_SSTAT1;
pSiop->pSstat0 = OFF_SSTAT0;
pSiop->pDstat = OFF_DSTAT;
pSiop->pDstatInt = OFF_DSTAT;
pSiop->pDs = OFF_DSA;
pSiop->pCtest3 = OFF_CTEST3;
pSiop->pCtest2 = OFF_CTEST2;
pSiop->pCtest1 = OFF_CTEST1;
pSiop->pCtest0 = OFF_CTEST0;
pSiop->pCtest6 = OFF_CTEST6;
pSiop->pCtest5 = OFF_CTEST5;
pSiop->pCtest4 = OFF_CTEST4;
pSiop->pTemp = OFF_TEMP;
pSiop->pIstat = OFF_ISTAT;
pSiop->pDfifo = OFF_DFIFO;
pSiop->pDCmd = OFF_DCMD;
pSiop->pDbc = OFF_DBC;
pSiop->pDnad = OFF_DNAD;
pSiop->pSp = OFF_DSP;
pSiop->pSpss = OFF_DSPS;
pSiop->pScratcha = OFF_SCRATCHA;
pSiop->pDcntl = OFF_DCNTL;
pSiop->pDwt = OFF_DWT;
pSiop->pDien = OFF_DIEN;
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
pSiop->pDmode = OFF_DMODE;
pSiop->pAdder = OFF_ADDER;
pSiop->pGpreg = OFF_GPREG;
pSiop->pSsid = OFF_SSID;
pSiop->pSist0 = OFF_SIST0;
pSiop->pSist1 = OFF_SIST1;
pSiop->pSlpar = OFF_SLPAR;
pSiop->pMacntl = OFF_MACNTL;
pSiop->pGpcntl = OFF_GPCNTL;
pSiop->pStime0 = OFF_STIME0;
pSiop->pStime1 = OFF_STIME1;
pSiop->pResp0 = OFF_RESP0;
pSiop->pRespid0 = OFF_RESPID0;
pSiop->pRespid1 = OFF_RESPID1;
pSiop->pStest0 = OFF_STEST0;
pSiop->pStest1 = OFF_STEST1;
pSiop->pStest2 = OFF_STEST2;
pSiop->pStest3 = OFF_STEST3;
pSiop->pScratchb = OFF_SCRATCHB;
```

```c
pSiop->scsiCtrl.clkPeriod = freqValue;
pSiop->scsiCtrl.maxBytesPerXfer = SIOP_MAX_XFER_LENGTH;
pSiop->scsiCtrl.scsiBusReset = (VOIDFUNCPTR) ncr810ScsiBusReset;
pSiop->scsiCtrl.scsiTransact = (FUNCPTR) ncr810ScsiTransact;
pSiop->scsiCtrl.scsiSyncMsgConvert = (FUNCPTR) ncr810SyncMsgConvert;
/* no longer needed with this kind of chip */
pSiop->scsiCtrl.scsiDevSelect = NULL;
pSiop->scsiCtrl.scsiBytesIn = NULL;
pSiop->scsiCtrl.scsiBytesOut = NULL;
pSiop->scsiCtrl.scsiDmaBytesIn = NULL;
pSiop->scsiCtrl.scsiDmaBytesOut = NULL;
pSiop->scsiCtrl.scsiBusPhaseGet = NULL;
pSiop->scsiCtrl.scsiMsgInAck = NULL;
pSiop->scsiCtrl.scsiSetAtn = NULL;
pSiop->slowCableMode = TRUE;
/* Just here to avoid a bus error in scsiAutoConfig
the chip can’t be programming for an another value (0 or 250ms) */
pSiop->scsiCtrl.scsiSelTimeOutCvt = (VOIDFUNCPTR) ncr810SelTimeOutCvt;
pSiop->pDevToSelect = (SCSI_PHYS_DEV *) NULL;
/* create the mutex siop data semaphore */
pSiop->pMutexData = semBCreate(SEM_Q_FIFO, SEM_FULL);
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    if (pSiop->pMutexData == (SEM_ID)NULL)
    {
        (void) cacheDmaFree ((char *) pSiop);
        (void) cacheDmaFree ((char *) pNcrCtxt);
        return((SIOP *)NULL);
    }

#ifndef NCR810_INT_LOG_MEM
    pNcr810Deb = (UINT *)malloc(LOG_SIZE_MEM);
    pLogMemShow = pNcr810Deb;
#endif
    return (pSiop);

/*******************************************************************************
* ncr810CtrlInit - initialize a control structure for the SIOP
* This routine initializes an SIOP structure, after the structure is created
* with ncr810CtrlCreate(). This structure must be initialized before the
* SIOP 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.
* Before using the SIOP, it must be initialized by calling this routine.
* Before returning, this routine pulses RST (reset) on the SCSI bus, thus
* resetting all attached devices.
* A detailed description of the input parameters follows:
* .iP <pSiop>
* a pointer to the NCR_810_SCSI_CTRL structure created with
* ncr810CtrlCreate().
* .iP <scsiCtrlBusId>
* the SCSI bus ID of the SIOP. Its value is somewhat arbitrary: seven (7),
* or highest priority, is conventional. The value must be in the range 0 - 7.
* .iP <scsiPriority>
* the priority to which a task is set when performing a SCSI transaction.
* Legal priorities are 0 to 255. Alternately, a -1 indicates that the priority
* should not be altered during SCSI transactions.
* RETURNS: OK, or ERROR if parameters are out of range.
*/

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

STATUS ncr810CtrlInit
{
    FAST NCR_810_SCSI_CTRL *pSiop, /* ptr to SIOP struct */
    FAST int scsiCtrlBusId, /* SCSI bus ID of this SIOP */
    int scsiPriority /* priority of a task when doing SCSI I/O */
}

int ix;
int jx;
NCR_CTL *pNcrCtl;
STATUS stat;
/* Check if pSiop not NULL */
if (pSiop == (SIOP *)NULL)
    return (ERROR);
pNcrCtl = &pSiop->pNcrCtxt->ncrCtl[0][0];
pSiop->pNcrCtl = pNcrCtl; /* added to avoid exceptions */
/* verify scsiCtrlBusId and enter legal value in SIOP structure */
if (scsiCtrlBusId < SCSI_MIN_BUS_ID || scsiCtrlBusId > SCSI_MAX_BUS_ID)
    return (ERROR);
pSiop->scsiCtrl.scsiCtrlBusId = (UINT8) scsiCtrlBusId;
/* translate id 0-7 in power of 2 (1000000 = 7). It’s for saving time *
 * at interrupt level in a reconnection scheme. */
pSiop->ctrlIdPow2 = (UINT8) (0x1 << pSiop->scsiCtrl.scsiCtrlBusId);
/* verify scsiPriority and enter legal value in SIOP structure */
if (scsiPriority < NONE || scsiPriority > 0xff)
    return (ERROR);
pSiop->scsiCtrl.scsiPriority = scsiPriority;
/* disconnect is supported for now, but this feature is not included *
 * in this revision. */
pSiop->scsiCtrl.disconnect = TRUE; /* Set TRUE, recompile to enable discon */
/* (Updated for 810 - bits now span 2 registers */
stat = NCR810_WRITE(pSiop->pSien0,1,0);
stat = NCR810_WRITE( pSiop->pSien1,1,0);
/* Added following - don’t want reset interrupt - jfr */
stat = NCR810_WRITE( pSiop->pSien0,1,B_MA|B_PAR|B_SGE);

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

ncr810ScsiBusReset (pSiop);
taskDelay(500);
/* initialize the SIOP hardware */
ncr810HwInit (pSiop);
stat = NCR810_WRITE( pSiop->pSien0,1,B_MA|B_PAR|B_SEL);
stat = NCR810_WRITE( pSiop->pSien1,1,B_STO);
/* Fill fixed values in the SIOP nexus */
for (ix = 0; ix <= SCSI_MAX_BUS_ID; ix++)
{
    for (jx = 0; jx <= SCSI_MAX_LUN; jx++)
    {
        /* Init fixed count */
        pNcrCtl->msgInCount = 1;
        pNcrCtl->extMsgInCount = 4;
        pNcrCtl->statusCount = 1;
        pNcrCtl->pIdentMsg = CACHE_DMA_VIRT_TO_PHYS (&pSiop->saveIdentMsg);
        pNcrCtl->identCount = 1;
        pNcrCtl++;
    }
}
return (OK);
}

/******************************************************************************
* ncr810ScsiBusReset - assert the RST line on the SCSI bus
* Issue a SCSI Bus Reset command to the NCR 810. This should put all devices
* on the SCSI bus in an initial quiescent state.
* This routine also clears the interrupt status by reading all status
* registers. Because the chip has an interrupt stack, it is necessary to
* loop on the status registers to clear any pending interrupts.
* RETURNS: N/A
* NOMANUAL
*/

/* LOCAL void ncr810ScsiBusReset */
void ncr810ScsiBusReset

(continued on next page)
Example 8-6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

{
    FAST SIOP *pSiop /* ptr to SIOP info */

    UINT tempReg;
    UINT tempReg1;
    STATUS stat;
    stat = NCR810_WRITE( pSiop->pScntl1,1,B_RST);
    taskDelay (2);
    stat = NCR810_READ(pSiop->pIstat,1,(void *)&tempReg); /* read Istat */
    stat = NCR810_WRITE( pSiop->pScntl1,1,0);
    /* flush the FIFO's */
    stat = NCR810_WRITE( pSiop->pCtest3,1,B_CLF);
    stat = NCR810_READ(pSiop->pSist0,1,(void *)&tempReg); /* read Sist0 */
    stat = NCR810_READ(pSiop->pSist0,1,(void *)&tempReg); /* read Sist0 */
    taskDelay (200);
    stat = NCR810_READ(pSiop->pSist1,1,(void *)&tempReg); /* read Sist1 */
    taskDelay (200);
    stat = NCR810_READ(pSiop->pDstat,1,(void *)&tempReg); /* read Dstat */
    taskDelay (200);

} /* ncr810Intr */

/******************************************************************************
 * ncr810Intr - interrupt service routine for the SIOP
 *
 * This interrupt routine just performs a clear of interrupt
 * conditions and copies all of the interrupt status of the SIOP
 * (status registers and script instruction interrupt values).
 * A check of the interrupt issued is done at the task level in
 * ncr810ScsiPhase function except for a phase mismatch, a new command to
 * start, and a reconnect process.
 *
 * RETURNS: N/A
 *
 * NOMANUAL
 */

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
void ncr810Intr
{
    SIOP *pSiop /* ptr to SIOP info */
}

UINT tmpReg;
UINT8 phaseRegReg;
TBOOL scriptIntr; /* used to detect true end in single step Mode */
UINT8 tmpLun; /* used to extract lun from ident msgin */
UINT8 tmp;
SCSI_PHYS_DEV *pScsiPhysDev; /* Used when reconnected occur */
UINT *localStart; /* Used to restart Script */
INT intrRegs = 0; /* local 32bit read of Dstat */
INT intrRegs1 = 0; /* local 32bit read of Sien */
STATUS stat,stat1;
int x;
ncr810CountIntr++;
/* Clear previous values */
pSiop->saveSist0 = 0;
pSiop->saveDstat = 0;
pSiop->saveScriptIntrStat=0;
/* Read interrupt status registers with a long acces to avoid any
* potential timing problem. The ncr810 Data manual specifys that a
* 128CLK period must be insert between two 8bit reads of SSTAT0
* and DSTAT. This limitation does not exist for a 32bit access.
*/
stat = NCR810_READ(pSiop->pIstat,1,(void *)&tmpReg); /* read Istat */
if (!tmpReg){ /* if bogus interrupt - exit */
    return;
}
/* The next 2 lines are for 32 bit reads so there is no timing problem */
if (tmpReg & (B_DIP)){ /* need to read sist0 and sist1 via sien0 */
    stat = NCR810_READ(pSiop->pDstatInt,4,(void *)&intrRegs); /* read Dstat */
}
if (tmpReg & (B_SIP)){ /* need to read sist0 and sist1 via sien0 */
    stat = NCR810_READ(pSiop->pSien0Int,4,(void *)&intrRegs1); /* read Sien */
}
pSiop->saveSist0 = (UINT8)((intrRegs1 >> 16) & 0xFF);
pSiop->saveSist1 = (UINT8)((intrRegs1 >> 24) & 0xFF);
pSiop->saveDstat = (UINT8)((intrRegs) & 0xFF);
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* check for Single Step interrupt */
if (pSiop->saveDstat & B_SSI)
    SCSI_INT_DEBUG_MSG("Single Step interrupt\n", 0, 0, 0, 0, 0, 0);

if (pSiop->saveDstat & B_SIR)
    scriptIntr = TRUE;

/* read the result value of int script instruction */
stat = NCR810_READ(pSiop->pDsps, 4, (void *)&tmpReg); /* read dsps */
pSiop->saveScriptIntrStat = tmpReg;

/* Process new command / disconnect / reconnected for exclusive
 access data part */
switch (pSiop->saveScriptIntrStat)
{
    case NEW_COMMAND_PROCESS:
        /* Initialize global nexus pointer */
        pSiop->pNcrCtl = pSiop->pNcrCtlCmd;
        #ifdef NCR810_INT_LOG_MEM
            LOG_MEM_INTR;
        #endif /* NCR810_INT_LOG_MEM */
        /* Flush all data involve in a transaction. */
        ncr810FlushCache(pSiop, pSiop->pNcrCtl);
        ncr810StartScript (pSiop, pSiop->pNcrCtl, 
                (SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev, 
                sysScsiConvertAddr((unsigned int *)pSiop->pNcrCtl->scriptAddr));
        return;
    case RECONNECT_PROCESS:
        SCSI_INT_DEBUG_MSG("ncr810Intr:reconnect\n", 0, 0, 0, 0, 0, 0);
        if ((pSiop->saveIdentMsg & SCSI_MSG_IDENTIFY) == 0)
        {
            /* It’s a bogus reconnect continue */
            logMsg ("ncr810Intr:improper reselect msgIn = 0x%02x\n", 
                    pSiop->saveIdentMsg, 0, 0, 0, 0, 0);
        }

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIO SCSI Driver Listing

```c
#ifdef NCR810_INT_LOG_MEM
LOG_MEM_INTR;
#endif /* NCR810_INT_LOG_MEM */
ncr810StartScript(pSiop, pSiop->pNcrCtl,
(SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev,
sysScsiConvertAddr((unsigned int *) &ackMsg1));
return;
}
tmpLun = pSiop->saveIdentMsg & SCSI_MAX_LUN;
/* Get scsi Id : we 've got the power of 2 both for
* own controller id and target id,xor first to get
* the target id  10000100 (ctr = 7 ,target = 2).
*/
/* Give devSyncSem for reconnected target by indexing ncrCtl
* device array.
*/
pScsiPhysDev = (SCSI_PHYS_DEV *)pSiop->pNcrCtlxt->
ncrCtl[tmpReg][tmpLun].pScsiPhysDev;
#endif /* NCR810_INT_LOG_MEM */

scsi810StartScript(pSiop, pSiop->pNcrCtl,
(SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev,
sysScsiConvertAddr((unsigned int *) &checkNewCmd));
return;
#endif /* NCR810_INT_LOG_MEM */
case RECONNECT_IN_SELECT:
#ifdef NCR810_INT_LOG_MEM
LOG_MEM_INTR;
#endif /* NCR810_INT_LOG_MEM */
    /* restart scsi phase routine for this device */
    semGive ((pScsiPhysDev->devSyncSem);
    return;
    case BAD_NEW_CMD:
#endif /* NCR810_INT_LOG_MEM */
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
ncr810StartScript(pSiop,pSiop->pNcrCtl,
    (SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev,
   sysScsiConvertAddr((unsigned int *)&waitSelect));
return;
default:
    break;
}
#endif /* NCR810_INT_LOG_MEM */
LOG_MEM_INTR;
#endif /* NCR810_INT_LOG_MEM */

/* handle the mismatch case at interrupt level because some count and
 * pointer have to be adjust with few chip register informations. */
if ((tmpReg = (pSiop->saveSist0 & B_MA)) == B_MA)
{
    localStart = ncr810PhaseMismatch(pSiop,
        (SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev,
        pSiop->pNcrCtl,(UINT)phaseReqReg);
nncr810StartScript(pSiop,pSiop->pNcrCtl,
    (SCSI_PHYS_DEV *)pSiop->pNcrCtl->pScsiPhysDev,
    sysScsiConvertAddr((unsigned int *) localStart));

    SCSI_INT_DEBUG_MSG("mismatch intr\n", 0, 0, 0, 0, 0, 0);
    return;
}
/* give the synchronize semaphore to any startscript caller
 * and check for single step feature. If single step is enable
 * check to give semaphore only if it an INT script opcode
 * occurred.
 */
if (siopSStep != TRUE)
{
    semGive (&pScsiPhysDev->devSyncSem);
}
else
{
    /* Give single step Semaphore */
    semGive (&pSiop->singleStepSem);
}
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
if (scriptIntr == TRUE)
{
    /* Give only if it’s an INT Script opcode executed */
    semGive (&pScsiPhysDev->devSyncSem);
}
}

/******************************************************************************
* ncr810HwInit - initialize the SIOP chip to a known state
* This routine puts the SIOP into a known quiescent state. Because the value
* of some registers are coupled with the hardware implemention of the chip you
* have to use ncr810SetHwRegister() to adjust the software/hardware couple.
* This routine does not touch the bits in register that are set by
* the ncr810SetHwRegister() routine.
* SEE ALSO
*  ncr810SetHwRegister()
*/

void ncr810HwInit
(
    SIOP *psIoP /* ptr to an SIOP info structure */
)
{
    unsigned int scratch;
    unsigned char temp;
    STATUS stat;
    /* Disable single step feature */
    siopSStep = FALSE;
    /* Perform a soft reset on Chip to get a known state */
    stat = NCR810_WRITE( pSiop->pIstat,1,B_SOFTRST);
    taskDelay(200); /* give it a chance to reset */
    stat = NCR810_WRITE( pSiop->pIstat,1,0);
    stat = NCR810_WRITE( pSiop->pDcntl,1,(B_COM | B_IRQM));
    stat = NCR810_WRITE( pSiop->pStime0,1,0xc); /* timeout time */
    stat = NCR810_WRITE( pSiop->pDien,1,0);
    temp = B_DHP; /* disable halt on parity error - for target mode only (really
does not apply */
}
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
if (pSiop->slowCableMode == TRUE)
    temp |= B_EXC; /* extra clock cycle of data setup */
stat = NCR810_WRITE( pSiop->pScntl1,1,temp);
stat = NCR810_WRITE( pSiop->pScid,1,(0x7 | B_SRE ));
stat = NCR810_WRITE( pSiop->pSi1en,1,(B_STO));
stat = NCR810_WRITE(pSiop->pSc10,1,(B_ARB1|B_ARB0|B_EPC|B_AAP|B_WATN));
/* set clock conversion factor, already checked in xxCtrlCreate */
scratch = pSiop->scsiCtrl.clkPeriod;
if ((scratch <= NCR810_1667MHZ) && (scratch >= NCR810_25MHZ)){
    stat = NCR810_WRITE( pSiop->pScntl3,1,(B_CCF0 | B_SCF0));
} else
    if ((scratch > NCR810_25MHZ) && (scratch <= NCR810_3750MHZ)){
        stat = NCR810_WRITE( pSiop->pScntl3,1,(B_CCF1 | B_SCF1));
    } else
        if ((scratch > NCR810_3750MHZ) && (scratch <= NCR810_50MHZ)){
            stat = NCR810_WRITE( pSiop->pScntl3,1,(B_CCF0|B_CCF1|B_SCF0|B_SCF1));
        } else
            if ((scratch > NCR810_50MHZ) && (scratch <= NCR810_6666MHZ)){
                stat = NCR810_WRITE( pSiop->pScntl3,1,(B_CCF2|B_SCF2));
            }
stat = NCR810_WRITE( pSiop->pDmode,1,(B_BL1));
stat = NCR810_WRITE( pSiop->pDi1en,1,(B_BF | B_AB1 | B_IID));
stat = NCR810_WRITE( pSiop->pStest3,1,(B_TE)); /* enable tolerant */
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/*******************************************************************************
* ncr810SetHwRegister - set Hardware dependant registers
* This routine is used to modify several registers default init values.
* Typically, this routine is called by the sysScsiInit() routine but is
* not required. The input parameters are the pointer
* returned by ncr810CtrlCreate() and a pointer to a NCR810_HW_REGS structure
* that is filled with the values for each register described below. This
* routine includes only registers that could modify the behavior of the chip.
* 
* .CS
* typedef struct
* { char scntl3;
*   char sxfer;
*   char ctest0;
*   char stime0;
* } NCR810_HW_REGS;
* 
* .CE
* To get a more detailed explanation regarding the description of each register
* involved see the User’s Manual of the NCR 53C810.
* 
* NOTE
* Because this routine writes to the chip registers you can’t use it
* if there is any SCSI bus activity.
* 
* RETURNS: OK or ERROR if any input parameter is NULL
* 
* SEE ALSO: ncr810.h, ncr810CtrlCreate()
* */
STATUS ncr810SetHwRegister
{
   SIOP *pSiop, /* Pointer to SIOP infos */
   NCR810_HW_REGS *pHwRegs /* Pointer to a NCR810_HW_REGS info */
}

{ STATUS stat;
  UINT tempReg;
  /* check input parameters */
  if ((pSiop == NULL) || (pHwRegs == NULL))
    return (ERROR);
  /* fill the SIOP structure with new parameters */

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
stat = NCR810_WRITE( pSiop->pScntl3,1,pHwRegs->scntl3); /* scntl3 */
stat = NCR810_WRITE( pSiop->pSxfer,1,pHwRegs->sxfer); /* sxfer */
stat = NCR810_WRITE( pSiop->pCtest0,1,pHwRegs->ctest0); /* ctest0 */
stat = NCR810_WRITE( pSiop->pStime0,1,pHwRegs->stime0); /* stime0 */

return (OK);
```

/*******************************************************************************
* ncr810ScsiPhase - Process a scsi transaction with the script code.
* This routine is called by ncr810ScsiTransact() to start the script
* that performs the scsi phase transaction. The ncrCtl shared data
* structure is filled with the apropriate parameters, pointer to the
* command block,message out area,local message in area,scsi status
* byte and data in/out buffer.
* The interrupt status is also processing in this routine. This can
* be done here because the mutex semaphore is taken in ncr810ScsiTransact.
* RETURNS: OK, or ERROR if not successful for any reason.
* 
* NOMANUAL
*/

```c
STATUS ncr810ScsiPhase
(
    SCSI_PHYS_DEV *pScsiPhysDev, /* ptr to the target device */
    SCSI_TRANSACTION *pScsiXaction /* ptr to the transaction info */
)
{
    ...
    IMPORT char extMsg[4];
    
    SCSI_DEBUG_MSG("ncr810Phase scsiAction=0x%0x -------\n", (int) pScsiXaction, 0, 0, 0, 0, 0);
    SCSI_DEBUG_MSG("pScsiPhysDev = 0x%08x", (int) pScsiPhysDev, 0, 0, 0, 0, 0);
}
```

(continued on next page)

8–84 Writing a SCSI Device Driver
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* controller info */
pScsiCtrl = pScsiPhysDev->pScsiCtrl;

pSiop = (SIOP *) (pScsiPhysDev->pScsiCtrl);

/* save current scsi command context requested */
pScsiPhysDev->pScsiXaction = pScsiXaction;

/* init current pointer to scsi info structure */
pSiop->pDevToSelect = pScsiPhysDev;

/* init current siop context structure */
  pNcrCtl = &pSiop->pNcrCtxt->ncrCtl
    [pScsiPhysDev->scsiDevBusId][pScsiPhysDev->scsiDevLUN];

/* If a disconnect will occur, retain physDev info pointer */
pNcrCtl->pScsiPhysDev = (UINT *)pScsiPhysDev;

/* initialise pointer in a shared data area pointer by pNcrCtl */
if (pScsiCtrl->scsiCtrlBusId == (UINT8)pScsiPhysDev->scsiDevBusId)
  return(ERROR);

/* Set target id in a data memory structure 00IDSX00 expects by the chip */
pNcrCtl->device = pScsiPhysDev->scsiDevBusId << 16;

stat = NCR810_READ(pSiop->pScntl3,1,(void *)&tempReg); /* read scntl3 */
pNcrCtl->device |= (tempReg << 24) ; /* Shift scntl3 into position */

/* Init Cmd count and pointer */
pNcrCtl->cmdCount = pScsiXaction->cmdLength;
pNcrCtl->pCmdLong = pScsiXaction->cmdAddress;
pNcrCtl->pCmd = sysScsiConvertAddr((unsigned int *)pNcrCtl->pCmdLong);

/* Init data count and pointer */
pNcrCtl->dataCount = pScsiXaction->dataLength;
pNcrCtl->pDataLong = pScsiXaction->dataAddress;
pNcrCtl->pData = sysScsiConvertAddr((unsigned int *)pNcrCtl->pDataLong);

/* set scsiPhysDev->savedTransLength, scsiPhysDev->savedTransAddress */
pScsiPhysDev->savedTransLength = pScsiXaction->dataLength;
pScsiPhysDev->savedTransAddress = pScsiXaction->dataAddress;

/* init message IN pointer. First save it for the 64 bit host side, then place in the PCI accessible address */
pNcrCtl->pMsgInLong = (UINT8 *) &msgIn[0];
pNcrCtl->pMsgIn = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgInLong);

pNcrCtl->pExtMsgInLong = (UINT8 *) &msgIn[1];
pNcrCtl->pExtMsgIn = sysScsiConvertAddr((unsigned int *)pNcrCtl->pExtMsgInLong);

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* init status address */
pNcrCtl->pScsiStatusLong = &pScsiXaction->statusByte;
pNcrCtl->pScsiStatus=sysScsiConvertAddr((unsigned int *)pNcrCtl->pScsiStatusLong);

/* Check if some message out have to be built. The useMsgout tag will
 * decide what kind of message has to be built. */
status = ncr810BuildMsgOut(pScsiPhysDev);
if (status == ERROR)
  return (ERROR);

/* Init message out count and pointer */
pNcrCtl->msgOutCount = pScsiPhysDev->msgLength;
pNcrCtl->pMsgOutLong = (UINT8 *) SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
pNcrCtl->pMsgOut = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);

/* choose the selWithAtn if useIdentify or/and useMsgout and build
 * Identify msgout. */
if ((pScsiPhysDev->useIdentify == TRUE) ||
   (pScsiPhysDev->useMsgout != SCSI_NO_MSGOUT)){
  pNcrCtl->scriptAddr = (unsigned int *) &selWithAtn;
}
else{
  pNcrCtl->scriptAddr = (unsigned int *) &selWithoutAtn;
}

if (pScsiPhysDev->useIdentify == TRUE)
  /* build identify message */
  pNcrCtl->pMsgOutLong = (UINT8 *) SCSI_GET_PTR_IDENTIFY(pScsiPhysDev);
pNcrCtl->pMsgOut=sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);
  identMsg = SCSI_MSG_IDENTIFY |
     (pSiop->scsiCtrl.disconnect ?SCSI_MSG_IDENT_DISCONNECT : 0) |
     (UINT8) pScsiPhysDev->scsiDevLUN;
PcrCtl->pMsgOutLong = identMsg;

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

if (pScsiPhysDev->useMsgout == SCSI_NO_MSGOUT)
    pNcrCtl->msgOutCount = 1;
else
    pNcrCtl->msgOutCount++;

/* Call the siopScriptPhase to perform a run on ncr chip */
SCSI_DEBUG_MSG("ncr810ScsiPhase:Calling FirstStartPhase \n", 0,0,0,0,0,0);
transactEnd = FALSE;

/* Set device status in SELECT_REQUESTED state */
pScsiPhysDev->devStatus = SELECT_REQUESTED;

status = ncr810FirstStartPhase (pSiop, pNcrCtl, pScsiPhysDev,
                        ncr810TimeOut);

if (status == ERROR)
    return(status);

while (transactEnd != TRUE)
{
    NCR810_DEBUG_MSG("siopIntrStatus=0x%08x", pSiop->saveScriptIntrStat);
    NCR810_DEBUG_MSG(" saveSist0=0x%02x", pSiop->saveSist0);
    NCR810_DEBUG_MSG(" saveDstat=0x%02x\n", pSiop->saveDstat);

    /* local pointer to msg out array */
    pMsgOut = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);

    if (((pSiop->saveDstat & B_SIR) != B_SIR)
    { /* Check for status registers interrupt */
        status = ncr810CheckStatRegs(pSiop, pScsiPhysDev,
                        &localStart);

        if (status == OK)
            goto contTransaction;

        /* all of these conditions are fatal */
        SCSI_DEBUG_MSG("out from phase with error\n", 0, 0, 0, 0, 0, 0);
        stat = NCR810_READ(pSiop->pSbcl,1,
                        (void *)&tempReg); /* read sbcl */
        SCSI_DEBUG_MSG("Scsi Phase End = 0x%02x\n",tempReg,
                        0, 0, 0, 0, 0, 0);
        goto exitPhase;
    }

    localSwitch = (pSiop->saveScriptIntrStat);

    (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* No error what's kind of intr script */
switch (localSwitch)
{
   case BAD_MSGIN_BEFORE_CMD:
   case BAD_MSG_AFTER_CMD:
   case BAD_MSG_AFTER_DATA:
   case BAD_MSG_INSTEAD_CMDCOMP:
      /* invalidate msgin array */
      CACHE_USER_INVALIDATE (pNcrCtl->pMsgInLong, MAX_MSG_IN_BYTES);
      LOG_ERR("Not an expected msg received!! msgin = 0x%02\n",
            msgIn[0]);
      LOG_ERR_CTX;
      /* Check if it's an ident message */
      if (msgIn[0] == identMsg)
         { /* Accept and restart to the next phase requested */
            if ((localSwitch == BAD_MSG_AFTER_DATA) ||
                (localSwitch == BAD_MSG_INSTEAD_CMDCOMP))
               localStart = (UINT *) &ackMsg3;
            else
               localStart = (UINT *) &ackMsg1;
         }
      else
         { /* reject Msg */
            LOG_ERR("Not an expected msg ..send reject msg!!\n");
            pNcrCtl->pMsgOutLong = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
            pNcrCtl->pMsgOut=sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);
            *pNcrCtl->pMsgOutLong = (UINT8)SCSI_MSG_MESSAGE_REJECT;
            pNcrCtl->msgOutCount = 1;
            localStart = (UINT *)&ackAttnMsg;
            /* flush msgout array */
            CACHE_USER_FLUSH (pNcrCtl->pMsgOutLong,1);
         }
      break;
   case MSGIN_EXPECT_AFTER_ST:
      /* not a msgIn phase after a status */
      LOG_ERR("msgIn expect after status!!\n");
      LOG_ERR_CTX;
      localStart = (UINT *) &checkPhData;
      break;
}

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOp SCSI Driver Listing

case MSGOUT_EXPECT:
    /* result of a select with atn and no msgout phase income, */
    /* goto the requested target phase. */
    
    LOG_ERR("Bad phase after select with atn!!\n");
    LOG_ERR_CTX;
    localStart = (UINT *) &asortPh;
    break;

case BAD_PH_AFTER_CMD:
    LOG_ERR("Bad phase after a command phase!!\n");
    LOG_ERR_CTX;
    /* it's not data/status/msgin ?, restart to the requested */
    /* phase. */
    
    localStart = (UINT *) &checkPhData;
    break;

case EXTMSG_BEFORE_CMD:
    case EXTMSG_AFTER_CMD:
    case EXTMSG_AFTER_DATA:
        LOG_ERR("Extended msg received!!\n");
        /* invalidate Msgin array */
        CACHE_USER_INVALIDATE (msgIn, sizeof(msgIn));
        /* read the remainder of the extended message using */
        /* mainline code here */
        
        localStart = ncr810ExtMsgRecv(pSiop, pScsiPhysDev, pNcrCtl,
                        &msgIn[0]);
        
        /* If we come back with error...the sync message parameters */
        /* were probably a mismatch...the following code will get the */
        /* chip running again */
        
        if (localStart == (UINT *) ERROR){
            ncr810ScsiBusReset(pSiop);
            ncr810HwInit(pSiop);
            ncr810ScriptStarted = FALSE;
            return ERROR;
        }
        
        /* Flush msgout array in case of reject msg */
        CACHE_USER_FLUSH (pNcrCtl->pMsgOutLong, MAX_MSG_OUT_BYTES);
        break;

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

case REJECT_MSG1:
    LOG_ERR("reject message received!!\n");
    LOG_ERR_CTX;
    localStart = ncr810RejectRecv(pSiop, pScsiPhysDev, pNcrCtl);
    /* flush msgout array */
    CACHE_USER_FLUSH (pNcrCtl->pMsgOutLong, MAX_MSG_OUT_BYTES);
    break;

case BAD_NEW_CMD:
    LOG_ERR("BAD NEW CMD!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;

case GOOD_END:
    transactEnd = (TBOOL) TRUE;
    break;

case ABORT_CLEAR_END:
    /* in this case return error ,because that means that you 
     * sent a message reset device and the target goes directly 
     * in bus free phase. 
     */
    LOG_ERR("ncr810ScsiPhase:abort end!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;

case BAD_PH_BEFORE_CMD:
    LOG_ERR("bad phase before cmd!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;

case PH_UNKNOWN:
    LOG_ERR("scsi phase unknown!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;

case RES_IN_DETECTED:
    case RES_OUT_DETECTED:
    LOG_ERR("RES4 RES5 phase detected!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;

(continued on next page)
case SAVDATP_BEFORE_CMD:
case SAVDATP_AFTER_CMD:
    NCR810_DEBUG_MSG("save pointer before the data phase\n");
    localStart = (UINT *) &ackMsg1;
    break;

case SAVDATP_AFTER_DATA:
    NCR810_DEBUG_MSG("save pointer after the data phase\n");
    /* Restart to accept message at a data phase point */
    localStart = (UINT *) &ackMsg3;
    break;

case DISC_BEFORE_CMD:
case DISC_AFTER_CMD:
case DISC_AFTER_DATA:
    NCR810_DEBUG_MSG("disconnect message\n");
    /* If it’s a first disconnection ,clear commandRequest.
    * The normal sequence is
    * IDLE->SELECT_REQUESTED->DISCONNECTED->RECONNECTED..->IDLE.
    */
    semTake(pSiop->pMutexData,WAIT_FOREVER);
    if (pScsiPhysDev->devStatus == SELECT_REQUESTED)
    {
        pSiop->commandRequest = FALSE;
    }
    if (pSiop->commandRequest == TRUE){
        stat = NCR810_READ(pSiop->pIstat,1,
            (void *)&tempReg); /* read Istat */
        tempReg |= B_SIGP;
        stat = NCR810_WRITE(pSiop->pIstat,1,tempReg);
    }
    semGive(pSiop->pMutexData);
    /* Save reconnect re-start script entry point */
    if (pSiop->saveScriptIntrStat == DISC_AFTER_DATA)
    localStart = (UINT *) &ackMsg3;
    else
    localStart = (UINT *) &ackMsg1;
    if (pScsiPhysDev->devStatus == SELECT_REQUESTED)
    {
        semGive(&pScsiCtrl->ctrlMutexSem);
    }

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* Update devStatus */
pScsiPhysDev->devStatus = DISCONNECTED;
/* Restart script to the idle point */
ncr810StartScript(pSiop, pSiop->pNcrCtl, pScsiPhysDev, sysScsiConvertAddr((unsigned int *)&waitSelect));
/* Suspend disconnected device */
semTake((pScsiPhysDev->devSyncSem, WAIT_FOREVER);
/* The reconnect will occur and restart the current SCSI transaction at this point for this device */
/* Restore nexus context */
pSiop->pNcrCtl = pNcrCtl;
/* Update dev status */
pScsiPhysDev->devStatus = RECONNECTED;
break;
case RESTORE_POINTER:
    NCR810_DEBUG_MSG("restore pointer received\n")
    LOG_ERR("restore pointer received\n");
pNcrCtl->pDataLong = pScsiPhysDev->savedTransAddress;
pNcrCtl->pData = sysScsiConvertAddr((unsigned int *)pNcrCtl->pDataLong);
pNcrCtl->dataCount = pScsiPhysDev->savedTransLength;
    localStart = (UINT *)&ackMsg1;
    break;
case SELECT_AS_TARGET:
    LOG_ERR("selected as a target!!\n");
    LOG_ERR_CTX;
    status = ERROR;
    goto exitPhase;
default:
    LOG_ERR("unknown intr script value!!\n");
    LOG_ERR("INT STATUS UNKNOWN:0x%08x pScsiPhysDev=0x%08x\n", pSiop->saveScriptIntrStat, pScsiPhysDev);
    LOG_ERR_CTX;
    status = ERROR;
    transactEnd = (TBOOL) TRUE;
    break;
}
contTransaction:
    /* transaction not yet finish restart script to next phase */

(continued on next page)
if (transactEnd == FALSE) {
    NCR810_DEBUG_MSG("calling startphase again\n");
    stat = NCR810_READ(pSiop->pScratcha,1,(void *)&tempReg); /* read Istat */
    status = ncr810StartPhase(pSiop, pNcrCtl, localStart, ncr810TimeOut);
}

exitPhase:
if (status == ERROR)
    NCR810_DEBUG_MSG("exit ERROR from Phase\n");
else
    NCR810_DEBUG_MSG("exit OK from Phase\n");
/* restore pending command context in case of reselect during
a select of a current starting */
semTake (pSiop->pMutexData,WAIT_FOREVER);
if ((pScsiPhysDev->devStatus == SELECT_REQUESTED) ||
(pScsiPhysDev->devStatus == SELECT_TIMEOUT))
pSiop->commandRequest = FALSE;
if (pSiop->commandRequest == TRUE){
    stat = NCR810_READ(pSiop->pIstat,1,(void *)&tempReg); /* read Istat */
    tempReg |= B_SIGP;
    stat = NCR810_WRITE( pSiop->pIstat,1,tempReg);
}
semGive(pSiop->pMutexData);
/* Give semaphore only if it’s not after a disconnect/reconnect */
if ((pScsiPhysDev->devStatus == SELECT_REQUESTED) ||
(pScsiPhysDev->devStatus == SELECT_TIMEOUT)){
    semGive (pScsiCtrl->ctrlMutexSem);
} /* invalidate or flush data */
if (pScsiPhysDev->pScsiXaction->dataDirection == O_WRONLY)
    CACHE_USER_FLUSH (pNcrCtl->pDataLong, pNcrCtl->dataCount);
else
    CACHE_USER_INVALIDATE (pNcrCtl->pDataLong, pNcrCtl->dataCount);
/* invalidate scsi status */
CACHE_USER_INVALIDATE (pNcrCtl->pScsiStatusLong, pNcrCtl->statusCount);
/* Restart Script at the Idle point */
ncr810StartScript (pSiop, pSiop->pNcrCtl, pScsiPhysDev,
sysScsiConvertAddr((unsigned int *) &waitSelect));

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    return (status);
}

/*******************************************************************************
* ncr810ScsiTransact - obtain exclusive use of SCSI controller for a transaction *
*
* This routine calls ncr810ScsiPhase() to execute the specified 
* command. If there are physical path management errors, then 
* this routine returns ERROR. If not, then the status returned 
* from the command is checked. If it is "Check Condition", then 
* a "Request Sense" CCS command is executed and the sense key is 
* examined. An indication of the success of the command is 
* returned (OK or ERROR). This routine is almost the same as the 
* scsiLib. If some modification occurs in the scsiLib.c, the same 
* change should be done here.
*
* RETURNS: OK if successful, or ERROR if a path management error occurs 
* or the status or sense information indicates an error.
*
* NOMANUAL
*/

LOCAL STATUS ncr810ScsiTransact
{
    SCSI_PHYS_DEV *pScsiPhysDev, /* ptr to the target device */
    SCSI_TRANSACTION *pScsiXaction /* ptr to the transaction info */
}

{  
    SCSI_CTRL *pScsiCtrl; /* SCSI controller info for device */
    SCSI_BLK_DEV_NODE *pBlkDevNode; /* ptr for looping through BLK_DEV list */
    STATUS status, stat; /* routine return status */
    int senseKey; /* extended sense key from target */
    int addSenseCode; /* additional sense code from target */
    long taskId = taskIdSelf (); /* calling task’s ID */
    int taskPriority; /* calling task’s current priority */

    pScsiCtrl = pScsiPhysDev->pScsiCtrl;
    /* set task priority to SCSI controller’s (if != NONE) */
    if (pScsiCtrl->scsiPriority != NONE)
    {
        taskPriorityGet (taskId, &taskPriority);
        taskPrioritySet (taskId, pScsiCtrl->scsiPriority);
    }
    /* take the device and controller semaphores */

    (continued on next page)
```
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    semTake (&pScsiPhysDev->devMutexSem, WAIT_FOREVER);
    if ((status = ncr810ScsiPhase(pScsiPhysDev, pScsiXaction)) == ERROR) {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: scsiPhaseSequence ERROR.\n", 0, 0, 0, 0, 0, 0);
        goto cleanExit;
    }

    /* check device status and take appropriate action */
    SCSI_DEBUG_MSG("ncr810ScsiTransact: Check Status value = 0x%x.\n", pScsiXaction->statusByte, 0, 0, 0, 0, 0);
    switch(pScsiXaction->statusByte & SCSI_STATUS_MASK) {
    case SCSI_STATUS_GOOD:
        status = OK;
        pScsiPhysDev->lastSenseKey = SCSI_SENSE_KEY_NO_SENSE;
        pScsiPhysDev->lastAddSenseCode = (UINT8) 0;
        goto cleanExit;
    case SCSI_STATUS_CHECK_CONDITION:
        {
            SCSI_COMMAND reqSenseCmd; /* REQUEST SENSE command */
            SCSI_TRANSACTION reqSenseXaction; /* REQUEST SENSE xaction */
            /* REQUEST SENSE buffer */
            UINT8 reqSenseData [REQ_SENSE_ADD_LENGTH_BYTE + 1];
            /* build a REQUEST SENSE command and transact it */
            (void) scsiCmdBuild (reqSenseCmd, &reqSenseXaction.cmdLength,
                               SCSI_OPCODE_REQUEST_SENSE,
                               pScsiPhysDev->scsiDevLUN, FALSE, 0,
                               pScsiPhysDev->reqSenseDataLength, (UINT8) 0);
            reqSenseXaction.cmdAddress = (UINT8 *) reqSenseCmd;
            /* if there is no user request sense buffer ,supply it */
            if (pScsiPhysDev->pReqSenseData == (UINT8 *)NULL)
                {
                    reqSenseXaction.dataAddress = reqSenseData;
    (continued on next page)
if (!pScsiPhysDev->extendedSense)
    reqSenseXaction.dataLength = 4;
else
    reqSenseXaction.dataLength = REQ_SENSE_ADD_LENGTH_BYTE + 1;
}
else{
    reqSenseXaction.dataAddress = pScsiPhysDev->pReqSenseData;
    reqSenseXaction.dataLength = pScsiPhysDev->reqSenseDataLength;
}

reqSenseXaction.dataDirection = O_RDONLY;
reqSenseXaction.addLengthByte = REQ_SENSE_ADD_LENGTH_BYTE;

SCSI_DEBUG_MSG("ncr810ScsiTransact: issuing a REQUEST SENSE command.
", 0, 0, 0, 0, 0, 0);

if ((status = ncr810ScsiPhase (pScsiPhysDev, &reqSenseXaction)) == ERROR)
    {
    SCSI_DEBUG_MSG("ncr810ScsiTransact: scsiPhaseSequence ERROR.
", 0, 0, 0, 0, 0, 0);
    goto cleanExit;
    }

/* REQUEST SENSE command status != GOOD indicates fatal error */
if (reqSenseXaction.statusByte != SCSI_STATUS_GOOD)
    {
    SCSI_DEBUG_MSG("scsiTransact: non-zero REQ SENSE status.
", 0, 0, 0, 0, 0, 0);
    errnoSet (S_scsiLib_REQ_SENSE_ERROR);
    status = ERROR;
    goto cleanExit;
    }

/* if device uses Nonextended Sense Data Format, return now */
if (!pScsiPhysDev->extendedSense)
    {
    status = ERROR;
    goto cleanExit;
    }

/* check sense key and take appropriate action */
pScsiPhysDev->lastSenseKey = (pScsiPhysDev->pReqSenseData)[2] & SCSI_SENSE_KEY_MASK;
pScsiPhysDev->lastAddSenseCode = (pScsiPhysDev->pReqSenseData)[12];

(continued on next page)
addSenseCode = (int) pScsiPhysDev->lastAddSenseCode;
switch (senseKey = (int) pScsiPhysDev->lastSenseKey)
{
    case SCSI_SENSE_KEY_NO_SENSE:
    {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: No Sense\n",
                       0, 0, 0, 0, 0);
        status = OK;
        goto cleanExit;
    }
    case SCSI_SENSE_KEY_RECOVERED_ERROR:
    {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: Recovered Error Sense,",
                       0, 0, 0, 0, 0);
        SCSI_DEBUG_MSG("Additional Sense Code = 0x%02x\n",
                       addSenseCode, 0, 0, 0, 0);
        status = OK;
        goto cleanExit;
    }
    case SCSI_SENSE_KEY_NOT_READY:
    {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: Not Ready Sense,",
                       0, 0, 0, 0, 0);
        SCSI_DEBUG_MSG("Additional Sense Code = 0x%02x\n",
                       addSenseCode, 0, 0, 0, 0);
        errnoSet (S_scsiLib_DEV_NOT_READY);
        status = ERROR;
        goto cleanExit;
    }
    case SCSI_SENSE_KEY_UNIT_ATTENTION:
    {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: Unit Attention Sense,",
                       0, 0, 0, 0, 0);
        SCSI_DEBUG_MSG("Additional Sense Code = 0x%02x\n",
                       addSenseCode, 0, 0, 0, 0);
        if (addSenseCode == 0x28)
        {
            semTake (&pScsiPhysDev->blkDevList.listMutexSem,
                      WAIT_FOREVER);
        }
}(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

for (pBlkDevNode = (SCSI_BLK_DEV_NODE *) lstFirst
    {pScsiPhysDev->blkDevList.blkDevNodes);
pBlkDevNode != NULL;
pBlkDevNode = (SCSI_BLK_DEV_NODE *)
    lstNext (&pBlkDevNode->blkDevNode))
{
pBlkDevNode->scsiBlkDev.blkDev.bd_readyChanged =
    TRUE;
}

semGive (&pScsiPhysDev->blkDevList.listMutexSem);
}
else if (addSenseCode == 0x29)
{
pScsiPhysDev->resetFlag = TRUE;
}

/* issue a MODE SENSE command */
{
    UINT8 modeSenseHeader [4];   /* mode sense data header */
    SCSI_COMMAND modeSenseCommand; /* SCSI command byte array */
    SCSI_TRANSACTION scsiXaction; /* info on a SCSI xaction */
    if (scsiCmdBuild (modeSenseCommand, &scsiXaction.cmdLength,
        SCSI_OPCODE_MODE_SENSE,
        pScsiPhysDev->scsiDevLUN, FALSE,
        0, sizeof (modeSenseHeader), (UINT8) 0)
        == ERROR)
        goto cleanExit;
    scsiXaction.cmdAddress   = modeSenseCommand;
    scsiXaction.dataAddress  = modeSenseHeader;
    scsiXaction.dataDirection = O_RDONLY;
    scsiXaction.dataLength   = sizeof (modeSenseHeader);
    scsiXaction.addLengthByte = MODE_SENSE_ADD_LENGTH_BYTE;

    SCSI_DEBUG_MSG("ncr810ScsiTransact: issuing a MODE SENSE cmd.\n",
        0, 0, 0, 0, 0, 0);
    if ((status = ncr810ScsiPhase (pScsiPhysDev,
        &scsiXaction)) == ERROR)
        goto cleanExit;
    SCSI_DEBUG_MSG("ncr810ScsiTransact returned ERROR\n",
        0, 0, 0, 0, 0, 0);
    goto cleanExit;
}

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    /* MODE SENSE command status != GOOD indicates 
     * fatal error 
     */

    if (scsiXaction.statusByte != SCSI_STATUS_GOOD)
        {
        SCSI_DEBUG_MSG("ncr810ScsiTransact: bad MODE SELECT stat.\n",
            0, 0, 0, 0, 0, 0);
        status = ERROR;
        goto cleanExit;
        }
    else
        {
        semTake (&pScsiPhysDev->blkDevList.listMutexSem, 
            WAIT_FOREVER);

        for (pBlkDevNode = (SCSI_BLK_DEV_NODE *)
            lstFirst 
            (&pScsiPhysDev->blkDevList.blkDevNodes);
            pBlkDevNode != NULL;
            pBlkDevNode = (SCSI_BLK_DEV_NODE *)
            lstNext (&pBlkDevNode->blkDevNode))
            {
            pBlkDevNode->scsiBlkDev.blkDev.bd_mode =
                (modeSenseHeader [2] & (UINT8) 0x80) ? O_RDONLY
                : O_RDWR;
            }

        semGive (&pScsiPhysDev->blkDevList.listMutexSem);

        (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    SCSI_DEBUG_MSG("Write-protect bit = \%x.\n", 
        (modeSenseHeader[2] & (UINT8) 0x80), 0, 0, 0, 0, 0);
    
    }  

    errnoSet (S_scsiLib_UNIT_ATTENTION);  
    status = ERROR;  
    goto cleanExit; 
    
    case SCSI_SENSE_KEY_DATA_PROTECT:  
    
        SCSI_DEBUG_MSG("ncr810ScsiTransact: Data Protect Sense,\n", 0, 0, 0, 0, 0, 0);  
        SCSI_DEBUG_MSG("Additional Sense Code = 0x\%02x\n",  
            addSenseCode, 0, 0, 0, 0, 0);  
        if (addSenseCode == 0x27)  
            errnoSet (S_scsiLib_WRITE_PROTECTED);  
            status = ERROR;  
            goto cleanExit; 
        
        default:  
            SCSI_DEBUG_MSG("ncr810ScsiTransact: Sense = \%x,\n",  
                senseKey, 0, 0, 0, 0, 0);  
            SCSI_DEBUG_MSG("Additional Sense Code = 0x\%02x\n",  
                addSenseCode, 0, 0, 0, 0, 0);  
            status = ERROR;  
            goto cleanExit; 
        
    }  

    case SCSI_STATUS_BUSY:  
        errnoSet (S_scsiLib_DEV_NOT_READY);  
        SCSI_DEBUG_MSG("device returned busy status.\n", 0, 0, 0, 0, 0, 0);  
        status = ERROR;  
        break;  

    default:  
        status = ERROR;  
        
    cleanExit:  
        /* give back controller and device mutual exclusion semaphores */  
        SCSI_DEBUG_MSG("ncr810ScsiTransact Give devMutexSem\n", 0, 0, 0, 0, 0, 0);  
        semGive (&pScsiPhysDev->devMutexSem);  

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIO PS Driver Listing

    /* restore task’s normal priority, if previously altered */
    if (pScsiCtrl->scsiPriority != NONE)
        taskPrioritySet (taskId, taskPriority);
    
    return (status);
}

/*******************************************************************************
* ncr810Show - Display values of all readable ncr810 (SIOP) registers *
* Displays the state of the SIOP registers in a user-friendly way. Primarily *
* used during debugging. The input parameter is the pointer to the SIOP info *
* structure returned by the ncr810CtrlCreate() call. *
* *
* NOTE
* The only readable register during a script execution is Istat register.
* If you use this routine during the execution of a SCSI comand the result *
* will be unpredictable.
* *
* EXAMPLE:
* .CS
* -> ncr810Show
* NCR810 Registers
* ----------------
* 0xfff47000: Sien = 0xa5 Sdid = 0x00 Scnt11 = 0x00 Scnt10 = 0x04
* 0xfff47004: Soc1 = 0x00 Sod1 = 0x00 Sxfer = 0x80 Scid = 0x80
* 0xfff47008: Sbc1 = 0x00 Sbd1 = 0x00 Sid1 = 0x00 Sfbr = 0x00
* 0xfff4700c: Sstat2 = 0x00 Sstat1 = 0x00 Sstat0 = 0x00 Dstat = 0x80
* 0xfff47010: Dsa = 0x00000000
* 0xfff47014: Ctest3 = ???? Ctest2 = 0x21 Ctest1 = 0xf0 Ctest0 = 0x00
* 0xfff47018: Ctest7 = 0x32 Ctest6 = ???? Ctest5 = 0x00 Ctest4 = 0x00
* 0xfff4701c: Temp = 0x00000000
* 0xfff47020: Lcrc = 0x00 Ctest8 = 0x00 Istat = 0x00 Dfifo = 0x00
* 0xfff47024: Dcmd/Ddc= 0x50000000
* 0xfff47028: Dnad = 0x00006614
* 0xfff4702c: Dsp = 0x00006614
* 0xfff47030: Dbps = 0x00006617
* 0xfff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
* 0xfff47038: Dcnt1 = 0x21 Dwt = 0x00 Dien = 0x37 Dmode = 0x01
* 0xfff4703c: Adder = 0x00000000
* value = 0 = 0x0
* .CE
* *
* SEE ALSO: ncr810CtrlCreate()
* *
* RETURNS: OK, or ERROR if <pScsiCtrl> and <pSysScsiCtrl> are both NULL.

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIO P SCSI Driver Listing

 STATUS ncr810Show
 { 
   FAST SCSI_CTRL *pScsiCtrl; /* ptr to SCSI controller info */
   }
   FAST SIOP *pSio;
   /* ptr to SIOP info */
   UINT tempReg;
   STATUS stat;
   if (pScsiCtrl == NULL)
   {
     if (pSysScsiCtrl == NULL)
     {
       printErr ("No SCSI controller specified. \n");
       return (ERROR);
     }
     pScsiCtrl = pSysScsiCtrl;
   }
   pSio = (SIOP *) pScsiCtrl;
   printf ("NCR810 Registers \n");
   printf ("-\-
");
   stat = NCR810_READ (pSio->pStime0, 1, (void *) &tempReg); /* Stime0 time */
   printf("Stime0 = 0x%02x ", tempReg);
   stat = NCR810_READ(pSio->pSien0, 1, (void *) &tempReg);
   printf("Sien0 = 0x%02x ", tempReg);
   stat = NCR810_READ(pSio->pSien1, 1, (void *) &tempReg);
   printf("Sien1 = 0x%02x ", tempReg);
   stat = NCR810_READ(pSio->pSdid, 1, (void *) &tempReg);
   printf("Sdid = 0x%02x ", tempReg);
   stat = NCR810_READ(pSio->pScntl1, 1, (void *) &tempReg);
   printf("Scntl0 = 0x%02x \n", tempReg);
   stat = NCR810_READ(pSio->pSocl, 1, (void *) &tempReg);
   printf("Socl = 0x%02x ", tempReg);
   (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
stat = NCR810_READ(pSiop->pSodl,1,
(void *)&tempReg);
printf ("Sodl = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSxfer,1,
(void *)&tempReg);
printf ("Sxfer = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pScid,1,
(void *)&tempReg);
printf ("Scid = 0x%02x\n ", tempReg);
stat = NCR810_READ(pSiop->pSbcl,1,
(void *)&tempReg);
printf ("Sbcl = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSbdl,1,
(void *)&tempReg);
printf ("Sbdl = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSidl,1,
(void *)&tempReg);
printf ("Sidl = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSfbr,1,
(void *)&tempReg);
printf ("Sfbr = 0x%02x\n ", tempReg);
stat = NCR810_READ(pSiop->pSstat2,1,
(void *)&tempReg);
printf ("Sstat2 = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSstat1,1,
(void *)&tempReg);
printf ("Sstat1 = 0x%02x ", tempReg);
stat = NCR810_READ(pSiop->pSstat0,1,
(void *)&tempReg);
printf ("Sstat0 = 0x%02x\n ", tempReg);
stat = NCR810_READ(pSiop->pDstat,1,
(void *)&tempReg);
printf ("Dstat = 0x%02x\n", tempReg);
stat = NCR810_READ(pSiop->pDsa,4,
(void *)&tempReg);
printf ("Dsa = 0x%08x\n", tempReg);
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
stat = NCR810_READ(pSiop->pCtest1,1,
                   (void *)&tempReg);
printf("Ctest1 = 0x%02x", tempReg);
stat = NCR810_READ(pSiop->pCtest2,1,
                   (void *)&tempReg);
printf("Ctest2 = 0x%02x", tempReg);
stat = NCR810_READ(pSiop->pCtest3,1,
                   (void *)&tempReg);
printf("Ctest3 = 0x%02x", tempReg);
stat = NCR810_READ(pSiop->pCtest0,1,
                   (void *)&tempReg);
printf("Ctest0 = 0x%02x\n", tempReg);
stat = NCR810_READ(pSiop->pCtest5,1,
                   (void *)&tempReg);
printf("Ctest5 = 0x%02x", tempReg);
stat = NCR810_READ(pSiop->pCtest4,1,
                   (void *)&tempReg);
printf("Ctest4 = 0x%02x\n", tempReg);
stat = NCR810_READ(pSiop->pTemp,1,
                   (void *)&tempReg);
printf("temp = 0x%08x\n", tempReg);
stat = NCR810_READ(pSiop->pIstat,1,
                   (void *)&tempReg);
printf("Istat = 0x%02x", tempReg);
stat = NCR810_READ(pSiop->pDfifo,1,
                   (void *)&tempReg);
printf("Dfifo = 0x%02x\n", tempReg);
stat = NCR810_READ(pSiop->pDbc,4,
                   (void *)&tempReg);
printf("Dcmd/Dbc = 0x%08x\n", tempReg);
stat = NCR810_READ(pSiop->pDnad,4,
                   (void *)&tempReg);
printf("Dnad = 0x%08x\n", tempReg);
stat = NCR810_READ(pSiop->pDsps,4,
                   (void *)&tempReg);
printf("Dsps = 0x%08x\n", tempReg);
stat = NCR810_READ(pSiop->pDcntl,1,
                   (void *)&tempReg);
printf("Dcntl = 0x%02x\n", tempReg);
```

(continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

    stat = NCR810_READ(pSiop->pDwt,1,
        (void *)&tempReg);
    printf ("Dwt = 0x%02x\n", tempReg);
    stat = NCR810_READ(pSiop->pDien,1,
        (void *)&tempReg);
    printf ("Dien = 0x%02x\n", tempReg);
    stat = NCR810_READ(pSiop->pDmode,1,
        (void *)&tempReg);
    printf ("Dmode = 0x%02x\n", tempReg);
    stat = NCR810_READ(pSiop->pAdder,1,
        (void *)&tempReg);
    printf ("Adder = 0x%08x\n", tempReg);
    stat = NCR810_READ(pSiop->pScntl3,1,
        (void *)&tempReg);
    printf ("scntl3 = 0x%02x\n", tempReg);
    stat = NCR810_READ(pSiop->pScntl2,1,
        (void *)&tempReg);
    printf ("scntl2 = 0x%02x\", tempReg);
    stat = NCR810_READ(pSiop->pScratcha,1,
        (void *)&tempReg);
    printf ("scratch = 0x%02x\", tempReg);
    return (OK);
}

/***********************************************************************
*  ncr810StartScript - Start a script code
*  Routine to start a script with the NCR chip. This routine takes
*  4 parameters, a SIOP pointer, the nexus pointer, the pScsiPhysdev pointer
*  and the script code entry point.  
*  <pScsiPhysDev> is used to set up the synchronous parameters if the target
*  supports this mode and the new ID descriptor is flushed. 
*  The Dsa register is load with the address of NCRTL (<pNcrCtl>) nexus data,
*  and the register Dsp with the address of the script routine to start.
*  Then the start bit is set in the Dcntl register. No mutex semaphores
*  are handled in this code. This routine is call from ncr810Intr.
*  
*  RETURNS: N/A
*  
*  NOMANUAL
*/

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
void ncr810StartScript
{
    SIOP *pSiop, /* pointer to SIOP info */
    NCR_CTL *pNcrCtl, /* ncr nexus info */
    SCSI_PHYS_DEV *pScsiPhysDev, /* scsi phys dev info */
    UINT scriptEntry /* physical routine address entry point */
}

    UINT tempReg;
    STATUS stat;
    unsigned int *temp; /* temp storage for kseg addr of scripts instruction */

    /* Check for Sync transfer init 00/ID/Xferp & offset/00 */
    if (pScsiPhysDev->syncXfer == TRUE)
    {
        /* init device descriptor with sxfer reg value */
        pNcrCtl->device |= pScsiPhysDev->syncReserved1;
        /* Add in the scntl3 sync value bits */
        pNcrCtl->device |= (pScsiPhysDev->syncReserved2 << 24);
    }
    else
    {
        /* Set assync mode */
        stat = NCR810_WRITE(pSiop->pSbcl,1,0);
        stat = NCR810_WRITE(pSiop->pSxfer,1,0);
        stat = NCR810_WRITE(pSiop->pScntl1,1,B_DHP);
    }

    /* Clear manual start bit */
    stat = NCR810_READ(pSiop->pDcntl,1,
        (void *)&tempReg); /* read dcntl */
    tempReg &= ~B_STD;
    stat = NCR810_WRITE(pSiop->pDcntl,1,tempReg);
    temp = CACHE_DMA_VIRT_TO_PHYS(pNcrCtl); /* get KSEG address */
    /* Use the sysScsiScripts Address to get a 32 bit physical address */
    stat = NCR810_WRITE(pSiop->pDsa,4,(UINT)sysScsiConvertAddr(temp));
    /* load data address */
}
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    /* Enable script interrupt */
    stat = NCR810_READ(pSiop->pDien,1,
                        (void *)&tempReg); /* read Dien */

    tempReg |= B_SIR;
    stat = NCR810_WRITE( pSiop->pDien,1,tempReg);
    /* load start address */
    stat = NCR810_WRITE( pSiop->pDsp,4,(UINT) scriptEntry);
}

/*******************************************************************************
* ncr810StartPhase - Start the script code phase
* This routine is use to restart a script to during a SCSI transaction after
* any script interrupt condition.
* The devSyncSem semaphore is taken and will be given by the interrupt routine.
* The return value is give by the script INT <value> instruction. If a
* semaphore timeout occurs during the transaction error status is returned.
* This timeout must be expressed in ticks.
* 
* RETURNS: Interrupt Status or ERROR
* 
* NOMANUAL
* */
LOCAL STATUS ncr810StartPhase
(
    SIOP *pSiop,       /* pointer to SIOP info */
    NCR_CTL *pNcrCtl,  /* script entry point */
    UINT *scriptAddress,  /* script entry point */
    int timeout /* Timeout on semTake */)
{

    FAST SCSI_PHYS_DEV *pScsiPhysDev = (SCSI_PHYS_DEV *)(pNcrCtl->pScsiPhysDev);

    NCR810_DEBUG_MSG("StartPhase pNcrCtl=0x%16x\n",pNcrCtl);
    NCR810_DEBUG_MSG("----- pScsiPhysDev=0x%16x\n",pScsiPhysDev);

    (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSiOP SCSI Driver Listing

ncr810StartScript (pSiop, pNcrCtl, pScsiPhysDev,
                   sysScsiConvertAddr((unsigned int *) scriptAddress));
if ((semTake ((pScsiPhysDev->devSyncSem,timeout) == ERROR))
{
    LOG_ERR("ncr810StartPhase : SemTake timeout\n");
    return (ERROR);
}
return (pSiop->saveScriptIntrStat);

/*******************************************************************************
* ncr810FirstStartPhase - Start the script code phase
* This routine starts the main script to handle a SCSI transaction. The
* controller semaphore <ctrlMutexSem> is taken and will be given
* by a disconnect or any command termination. The devSyncSem semaphore is
* given at interrupt level. The pMutexData semaphore is used to set up the global information needed
* to start the new command at interrupt level. It is used to start the first select for a SCSI transaction. The SIGP is set
* to start the current command by aborting the <wait SELECT..> script
* instruction at interrupt level. The return value is given by the script INT <value> instruction.
* If a semaphore timeout occurs during the transaction an ERROR status is
* returned. This timeout must be expressed in tick.
* RETURNS: Script Interrupt Status or ERROR
* NOMANUAL
*/
STATUS ncr810FirstStartPhase
{
    SIOP *pSiop,          /* pointer to SIOP info */
    NCR_CTL *pNcrCtl,     /* pointer to siop nexus */
    SCSI_PHYS_DEV *pScsiPhysDev,    /* pointer to device info */
    int timeout             /* Timeout on semTake */
}
{
    SCSI_CTRL *pScsiCtrl = (SCSI_CTRL *)pSiop;
    STATUS stat;
    UINT tempReg;
    /* I added the following here because I couldn’t call it from ncr810HwInit
    above (see comment) - jfr */

    (continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

```c
semTake (&pScsiCtrl->ctrlMutexSem, WAIT_FOREVER);
semTake (pSiop->pMutexData, WAIT_FOREVER);
if (ncr810ScriptStarted == FALSE) {
    ncr810StartScript (pSiop, pNcrCtl, pScsiPhysDev,
                     sysScsiConvertAddr((unsigned int *) &waitSelect));
    ncr810ScriptStarted = TRUE;
taskDelay(200);
}
pSiop->pNcrCtlCmd = pNcrCtl;
/* Set sigp bit to start the command at interrupt level */
pSiop->commandRequest = TRUE;
stat = NCR810_READ(pSiop->pIstat, 1,
                   (void *)&tempReg); /* read Istat */
    tempReg |= B_SIGP;
stat = NCR810_WRITE( pSiop->pIstat, 1, tempReg);
semGive(pSiop->pMutexData);
if ((semTake (pScsiPhysDev->devSyncSem, timeOut) == ERROR))
    {LOG_ERR("ncr810StartFirstPhase : SemTake timeout\n");
        return (ERROR);
    }
return (pSiop->saveScriptIntrStat);
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/*******************************************************************************
* ncr810SyncMsgConvert - Compute the ncr810 values for synchronous tranfert *
* This routine should be called when an incoming extended synchronous message *
* has been read. This routine will try to find out the closest value possible *
* for REQ/ACK period to fit the contents of the target message value. *
* If it’s impossible to fit exactly, it returns the closest value it finds. *
* If the input value is too slow (<0xfa async throughput) or too high (>0x19 *
* around 100MBit/s) a null offset and period will be return. *
* If the offset is greater than the maximum chip value, it’s set to the *
* maximum offset allowable for the chip. *
* If some of input parameters are not consistant an ERROR will be returned. *
* The hardware register (SBCL and SXFER) values are computed and returned *
* in the syncReserved1 and synvReserved2 field in the SCSI_PHYS_DEV structure *
* to be used each time by the driver when the SCSI_PHYS_DEV have a <syncXfer> *
* field set to TRUE. *
* RETURNS: OK or ERROR if some bad parameters. *
* NOTE: As this routine uses input clock frequency and clock division, *
* it should be called after a ncr810CtrlCreate and ncr810CtrlInit. *
* SEE ALSO *
* scsiSyncTarget(), scsiLib, scsiLib.h *
* NOMANUAL *
*******************************************************************************/

STATUS ncr810SyncMsgConvert
{
    SCSI_PHYS_DEV *pScsiPhysDev, /* ptr to device info */
    UINT8 *pMsgIn, /* ptr to extended message */
    int *pSyncVal, /* to return sync info */
    int *pOffsetVal /* to return offset info */

    SIOP *pSiop; /* local pointer to the controller info */
    int coreClock; /* core clock value */
    int div; /* core clock divider */
    int bestFound;
    int bestXferp;
    int bestDiv;
    int xferLoop;
    int periodValue;
    int offsetValue;
    int tmp = 0 ;

    (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    if ((pScsiPhysDev == (SCSI_PHYS_DEV *) NULL) ||
        (pMsgIn == NULL) ||
        (pSyncVal == NULL) ||
        (pOffsetVal == NULL)) {
        return (ERROR);
    }

    /* Get period and offset from the incoming message */
    periodValue = (int)pMsgIn[3];
    offsetValue = (int)pMsgIn[4];

    if ((periodValue > MAX_SCSI_SYNC_PERIOD) ||
        (periodValue < MIN_SCSI_SYNC_PERIOD)) {
        errnoSet (S_scsiLib_SYNC_VAL_UNSUPPORTED);
        return (ERROR);
    }

    /* check for a chip supported offset value */
    if ((UINT)offsetValue > (UINT)NCR810_MAXSYNC_OFF)
        offsetValue = NCR810_MAXSYNC_OFF;

    /* express period in ns * 4 (SCSI ANSI) and multiply by 100 to
     * have a fix decimal part. */
    periodValue = periodValue * SCSI_SYNC_STEP_PERIOD * 100;
    pSiop = (SIOP *) (pScsiPhysDev->pScsiCtrl);
    bestFound = MAX_SCSI_SYNC_PERIOD * SCSI_SYNC_STEP_PERIOD * 100;
    bestXferp = NCR810_MAX_XFERP + 1;
    bestDiv = 0;

    coreClock = pSiop->scsiCtrl.clkPeriod;

    /* Find out the closest division value to match the scsi period requested. */
    /* The formulas used are : */
    /* PERIOD = TCP * (4 + XFERP) */
    /* TCP = 1/coreClock */
    /* XFERP is the sync transfer period code used by the */
    /* chip (range 0-7). */
    /* coreClock is the scsi core clock input divided by 1, 1.5, 2. */
    /* or coreClock * (2/2), (2/3), (2/4). */
    /* */
    for (div = 1; div < 4; div++) {
        for (xferLoop = 0; xferLoop <= 7; xferLoop++) {
            tmp = ((coreClock * (div+1)) * (4 + xferLoop)) / 2;
            (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
if (tmp >= periodValue)
{
    if ((bestFound) > tmp)
    {
        bestFound = tmp;
        bestXferp = xferLoop;
        bestDiv = div;
    }
}
}

if (bestXferp > NCR810_MAX_XFERP)
{
    /* nothing fits properly use async tranfert */
    *pSyncVal = periodValue / (SCSI_SYNC_STEP_PERIOD * 100);
    *pOffsetVal = 0;
    return (OK);
}

/* Xfer register values */
pScsiPhysDev->syncReserved1 = (offsetValue * 256);
pScsiPhysDev->syncReserved1 |= (bestXferp * 4096);
/* sbcl register clock divider value */
pScsiPhysDev->syncReserved2 = (UINT8) bestDiv;
/* return computed values */
*pSyncVal = bestFound / (SCSI_SYNC_STEP_PERIOD * 100);
*pOffsetVal = offsetValue;
return(OK);
}
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/*******************************************************************************
* ncr810BuildMsgOut - Build a message out
* *
* Build a message out in the scsiPhysDev structure according to the
* <useMsgOut> code in the scsiPhysDev structure.
* Only two message type are supported, synchronous and device reset message.
* The value code <SCSI_OTHER_MSGOUT> allow the user to build his own message
* to send to the target.
* For the synchronous message the specific driver routine is called. If the
* Driver do not support currently this feature a <S_scsiLib_SYNC_UNSUPPORTED>
* is returned.
* Because the message count is not incremental only one message should be
* build at time.
* *
* RETURNS: ERROR or OK
* *
* NOMANUAL
* */

LOCAL STATUS ncr810BuildMsgOut
{
    SCSI_PHYS_DEV *pScsiPhysDev /* pointer to phys dev info */

    UINT8 *pMsgOut;
    STATUS status;
    int syncVal;
    int offsetVal;

    switch (pScsiPhysDev->useMsgout)
    {
        case SCSI_OTHER_MSGOUT:
            /* this is a user message ,don’t modify anything */
            break;

        case SCSI_NO_MSGOUT:
            /* no message out to build clear msgLenght */
            pScsiPhysDev->msgLength = 0;
            break;

        case SCSI_WIDE_MSGOUT:
            /* Not supported */
            pScsiPhysDev->msgLength = 0;
            break;
    }

    (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

case SCSI_DEV_RST_MSGOUT:
    pMsgOut = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
pMsgOut[0] = SCSI_DEV_RST_MSGOUT;
pScsiPhysDev->msgLength = 1;
    break;

case SCSI_SYNC_MSGOUT:
    /* Check if the driver is supporting sync capability */
    NCR810_DEBUG_MSG("SCSI_SYNC_MSGOUT\n");
    if (pScsiPhysDev->pScsiCtrl->scsiSyncMsgConvert == (FUNCPTR) NULL) {
        errnoSet(S_scsiLib_SYNC_UNSUPPORTED);
        return(ERROR);
    }
    /* Build Message out */
    syncVal = pScsiPhysDev->syncXferPeriod / SCSI_SYNC_STEP_PERIOD;
    offsetVal = pScsiPhysDev->syncXferOffset;
pMsgOut = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
    status = scsiBuildExtMsgOut(pMsgOut, SCSI_EXT_MSG_SYNC_XFER_REQ);
    /* set sync Value in extended sync message */
    if ((syncVal < MIN_SCSI_SYNC_PERIOD) ||
        (syncVal > MAX_SCSI_SYNC_PERIOD))
        syncVal = MIN_SCSI_SYNC_PERIOD;
    /* round up value for the controller */
    pMsgOut[4] = (UINT8) syncVal;
pMsgOut[5] = (UINT8) offsetVal;
    NCR810_DEBUG_MSG("msgout[0]=0x%02x\n", pMsgOut[0]);
    NCR810_DEBUG_MSG("msgout[1]=0x%02x\n", pMsgOut[1]);
    NCR810_DEBUG_MSG("msgout[2]=0x%02x\n", pMsgOut[2]);
    NCR810_DEBUG_MSG("msgout[3]=0x%02x\n", pMsgOut[3]);
    NCR810_DEBUG_MSG("msgout[4]=0x%02x\n", pMsgOut[4]);
    NCR810_DEBUG_MSG("syncVal=0x%02x", syncVal);
    NCR810_DEBUG_MSG("offsetVal=0x%02x\n", offsetVal);
    if (status == ERROR)
        return(ERROR);
    /* set the round up values in the message out */
    pMsgOut[4] = (UINT8) syncVal;
pMsgOut[5] = (UINT8) offsetVal;
    /* Set Message Count */

(continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

```c
pScsiPhysDev->msgLength = 0x6;
break;

default:
return(ERROR);
}
return(OK);

/******************************************************************************
*  ncr810PhaseMismatch - Handle a phase mismatch case.
*  It handles the phase mismatch case on the scsi. This is occurred when
*  a target plans to disconnect during any data and message transfer phase or
*  when the device is not able to read more data (Tape device).
*  This routine updates the data or message count according to the state of
*  the ncr fifo and internal registers. The save data pointers are also
*  updated to avoid any copy if a disconnect occur.
*  The current phase requested is determined from the <phaseRequested>
*  parameter.
*  RETURNS:Script address to execute.
*  NOTE:This routine is called at interrupt level.
*  NOMANUAL
*/

LOCAL UINT *ncr810PhaseMismatch
```
```c
    (SIOP *pSiop, /* pointer to controller info */
    SCSI_PHYS_DEV *pScsiPhysDev, /* pointer to phys dev info */
    NCR_CTL *pNcrCtl, /* pointer to the current siop context */
    UINT phaseRequested /* phase requested */
    )
```
```c
    {
    UINT ncrRegCount;
    UINT8 tmpCount;
    UINT countFifo;
    UINT *localStart;
    STATUS stat;
    UINT tempReg;
    NCR810_MISMATCH_MSG("scsi phase mismatch error\n", 0, 0, 0, 0, 0, 0);
```

(continued on next page)
/* Clear any msg out request */
pscildPhysDev->useMsgout = SCSI_NO_MSGOUT;

/* update pointer for data in if there is some data in sidl register */
stat = NCR810_READ(pSiop->pDbc,1,
    (void *)&ncrRegCount); /* read dbc */
ncrRegCount &= NCR810_COUNT_MASK;
stat = NCR810_READ(pSiop->pDbc,4,
    (void *)&tempReg); /* read dbc */
NCR810_MISMATCH_MSG("ncr DBC = 0x%08x ", tempReg, 0, 0, 0, 0);
NCR810_MISMATCH_MSG("pData=0x%08x count=0x%08x\n", (int) pNcrCtl->pDataLong,
    pNcrCtl->dataCount, 0, 0, 0, 0);

/* Check phase requested to update the good couple pointer/count */
switch (phaseRequested)
{
    case PHASE_DATAIN:
    case PHASE_MSGIN:
        stat = NCR810_READ(pSiop->pSstat1,1,
            (void *)&tempReg); /* read stat1 */
        if (tempReg & B_ILF)
            { /* update user buffer */
                stat = NCR810_READ(pSiop->pSidl,1,
                    (void *)&tempReg); /* read sidl */
                stat = NCR810_WRITE( pSiop->pDnad,4,tempReg);
                ncrRegCount--;
            }
        stat = NCR810_READ(pSiop->pCtest3,1,
            (void *)&tempReg); /* read Ctest3 */
        tempReg |= B_CLF;
        stat = NCR810_WRITE( pSiop->pCtest3,1,tempReg);
    /* Update data pointer and count */
    if (phaseRequested == PHASE_MSGIN)
        { (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
pNcrCtl->pMsgInLong += (pNcrCtl->msgInCount - ncrRegCount);

pNcrCtl->pMsgIn = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgInLong);

pNcrCtl->msgInCount -= (pNcrCtl->msgInCount - ncrRegCount);

NCR810_MISMATCH_MSG("mismatch msgin\n", 0, 0, 0, 0, 0, 0);

localStart = (UINT *) &ackMsg1;

else
{
pNcrCtl->pDataLong += (pNcrCtl->dataCount - ncrRegCount);

pNcrCtl->pData = sysScsiConvertAddr((unsigned int *)pNcrCtl->pDataLong);

pNcrCtl->dataCount -= (pNcrCtl->dataCount - ncrRegCount);

pScsiPhysDev->savedTransLength = pNcrCtl->dataCount;

pScsiPhysDev->savedTransAddress = pNcrCtl->pDataLong;

NCR810_MISMATCH_MSG("mismatch datain ", 0, 0, 0, 0, 0, 0);

NCR810_MISMATCH_MSG("pData=0x%08x count=0x%08x\n",
                   (int) pNcrCtl->pDataLong,pNcrCtl->dataCount,
                   0, 0, 0, 0);

localStart = (UINT *)&checkPhData;
}

/* Clear Fifo and restart where phase is checked */

stat = NCR810_READ(pSiop->pCtest3,1,
                   (void *)&tempReg); /* read Ctest3 */

    tempReg |= B_CLF;

    stat = NCR810_WRITE( pSiop->pCtest3,1,tempReg);

    return (localStart);

case PHASE_DATAOUT:
    case PHASE_MSGOUT:
    
    /* Check for data in fifo and registers (SODR & SODL),
     * count from DCMD minus DFIFO count and 0x7f (see ncr810
     * data manuel).
     */

    NCR810_MISMATCH_MSG("mismatch dataout\n", 0, 0, 0, 0, 0);

    tmpCount = (UINT8)(ncrRegCount & 0x7f);

    stat = NCR810_READ(pSiop->pDfifo,1,
                       (void *)&countFifo); /* read Ctest3 */

    countFifo &= 0x7f;

    tmpCount = (countFifo - tmpCount) & 0x7f;

    /* Check for SODL and SODR if residual data */
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
stat = NCR810_READ(pSiop->pSstat0,1,
    (void *)&tempReg); /* read sstat0 */

if (tempReg & B_OLF)  
tmpCount++;  
if (tempReg & B_ORF)  
tmpCount++;  
ncrRegCount += tmpCount;
/* Update dataCount and pointer */
if (phaseRequested == PHASE_MSGOUT)  
{
    pNcrCtl->pMsgOutLong += (pNcrCtl->msgOutCount - ncrRegCount);
pNcrCtl->pMsgOut = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);
pNcrCtl->msgOutCount -= (pNcrCtl->msgOutCount - ncrRegCount);
    localStart = (UINT *) &asortPh;
    NCR810_MISMATCH_MSG("mismatch msgout\n", 0, 0, 0, 0, 0, 0);
}
else  
{
    pNcrCtl->pDataLong += (pNcrCtl->dataCount - ncrRegCount);
pNcrCtl->pData = sysScsiConvertAddr((unsigned int *)pNcrCtl->pDataLong);
pNcrCtl->dataCount -= (pNcrCtl->dataCount - ncrRegCount);
pScsiPhysDev->savedTransLength = pNcrCtl->dataCount;
pScsiPhysDev->savedTransAddress = pNcrCtl->pDataLong;
    localStart = (UINT *) &checkPhData;
    NCR810_MISMATCH_MSG("mismatch dataout ", 0, 0, 0, 0, 0, 0);
    NCR810_MISMATCH_MSG("pData=0x%08x count=0x%08x\n", 
        (int) pNcrCtl->pDataLong, pNcrCtl->dataCount, 0, 0, 0, 0);
}
/* Clear Fifo and restart where phase is checked */
stat = NCR810_READ(pSiop->pCtest3,1,
    (void *)&tempReg); /* read Ctest3 */

    tempReg |= B_CLF;
    stat = NCR810_WRITE( pSiop->pCtest3,1,tempReg);
/* *pSiop->pCtest3|= B_CLF; */ /* now on ctest3 */
return(localStart);
```

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

default:
    NCR810_MISMATCH_MSG("mismatch default unknown phase\n", 0, 0, 0, 0, 0, 0);
    /* clear ncr fifo */
    stat = NCR810_READ(pSiop->pCtest3,1, (void *)&tempReg); /* read Ctest3 */
    tempReg |= B_CLF;
    stat = NCR810_WRITE(pSiop->pCtest3,1,tempReg);
    return ((UINT *) &asortPh);
}

/*******************************************************************************
  * ncr810ExtMsgRecv - Handle an extended message received.
  *
  * It check the type of extended message received. If it’s not supported a
  * message reject will be built. Only the sync extended message is supported.
  * If the synchronous target message values can’t fit with the controller,
  * a reset device message is sent to the target to be sure to get a
  * asynchronous protocol.
  *
  * RETURNS: Script address to execute.
  *
  * NOMANUAL
  *
  */

UINT *ncr810ExtMsgRecv
{
    SIOP *pSiop, /* pointer to controller info */
    SCSI_PHYS_DEV *pScsiPhysDev, /* pointer to phys dev info */
    NCR_CTL *pNcrCtl, /* pointer to the current siop context */
    UINT8 *msgIn /* pointer to msgin array */

    int syncVal;
    int offsetVal;
    UINT *localStart;
    STATUS status;
    UINT tempReg;
    STATUS stat;

    (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

LOG_ERR("msgin[0]=0x%02x
", msgIn[0], 0, 0, 0, 0, 0);
LOG_ERR("msgin[1]=0x%02x
", msgIn[1], 0, 0, 0, 0, 0);
LOG_ERR("msgin[2]=0x%02x
", msgIn[2], 0, 0, 0, 0, 0);
LOG_ERR("msgin[3]=0x%02x
", msgIn[3], 0, 0, 0, 0, 0);
LOG_ERR("msgin[4]=0x%02x
", msgIn[4], 0, 0, 0, 0, 0);

switch (msgIn[2])
{
    case SCSI_EXT_MSG_SYNC_XFER_REQ:
        /* Find out hardware value for this chip to try to fit the *
         * target value.
        */
        status = ncr810SyncMsgConvert((SCSI_PHYS_DEV *)pNcrCtl->pScsiPhysDev,
                                      msgIn, &syncVal, &offsetVal);
        if ((syncVal != msgIn[3]) || (syncVal == 0) ||
            (offsetVal == 0) || (status == ERROR))
        {
            /* use msg out array build a reset msg */
            logMsg("\nCorrect period/offset for scsiSyncTarget() should be %d/%d\n",
                   syncVal,msgIn[4],0,0,0,0);
            /* Use SCSI_PHYS_DEV structure for the value to be returned *
             * by the target to the caller, it’s possible because the *
             * <pScsiPhysDev->syncXfer> is used to check if the device *
             * supports the sync transfer. *
             */
            pScsiPhysDev->syncXferPeriod = syncVal * 4;
            pScsiPhysDev->syncXferOffset = msgIn[4];
            pScsiPhysDev->syncXfer = (TBOOL) FALSE;
            pNcrCtl->pMsgOutLong = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
            pNcrCtl->pMsgOut = sysScsiConvertAddr((unsigned int *) pNcrCtl->pMsgOutLong);
            *pNcrCtl->pMsgOutLong = (UINT8) SCSI_MSG_BUS_DEVICE_RESET;
            pNcrCtl->msgOutCount = 1;
            return ( (UINT *) ERROR);
        }

        stat = NCR810_READ(pSiop->pScntl1,1,
                (void *)&tempReg); /* read Scntl1*/
        tempReg &= B_DHP; /* Why AND? */
        /* Set up sync values in physDev info for future use */
        /* (continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

    pScsiPhysDev->syncXferOffset = msgIn[4];
    /* set sync tranfert to true if the offset is not null
     * (async agreement). */
    if (msgIn[4] > 0)
    {
        pScsiPhysDev->syncXfer = (TBOOL) TRUE;
    }
    localStart = (UINT *) &contTransact;
    /* One more clr ack is necessary */
    stat = NCR810_READ(pSiop->pSocl,1, (void *)&tempReg);
    tempReg &= 0xbf; /* clear ack */
    stat = NCR810_WRITE( pSiop->pSocl,1,tempReg);
    break;

default:
    /* not a supported message send a reject Message */
    LOG_ERR("unsupported extended message\n");
    pNcrCtl->pMsgOutLong = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
    pNcrCtl->pMsgOut = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);
    *pNcrCtl->pMsgOutLong = (UINT8)SCSI_MSG_MESSAGE_REJECT;
    pNcrCtl->msgOutCount = 1;
    localStart = (UINT *)&ackAttnMsg;
    break;
}
return(localStart);

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/*******************************************************************************
* ncr810RejectRecv - Handle a reject message received.
*
* Check what the last message sent to determine the appropriate action.
* If it’s a sync extended message all the sync capability both for the hardware
* and the physDev structure will be cleared.
* If it’s was a reject message or anything else, send a NO OP message to the
* target.
*  
* RETURNS: Script address to execute.
*  
* NOMANUAL
*  
*/

LOCAL UINT *ncr810RejectRecv

  {
    SIOP *pSiop,          /* pointer to controller info */
    SCSI_PHYS_DEV *pScsiPhysDev, /* pointer to phys dev info */
    NCR_CTL *pNcrCtl     /* pointer to the nexus context */
  }

  |
  STATUS stat;
  UINT tempReg;
  |
  /* if the previous msg was a sync msg, means that the target does not
  * support the synchronous protocol. */
  pNcrCtl->pMsgOutLong = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
  pNcrCtl->pMsgOut = sysScsiConvertAddr((unsigned int *)pNcrCtl->pMsgOutLong);
  if ((pNcrCtl->pMsgOutLong[0] == SCSI_MSG_EXTENDED_MESSAGE) &&
  {
    stat = NCR810_WRITE( pSiop->pSbcl,1,tempReg);
    /* Clear Hardware sync register */
    stat = NCR810_READ(pSiop->pScntl1,1,
      (void *)&tempReg); /* read Scntl1*/
    tempReg &= B_DHP; /* Why AND? */
    stat = NCR810_WRITE( pSiop->pScntl1,1,tempReg);
  }

  (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

    /* Clear sync capability in physDev structure */
    pScsiPhysDev->syncXfer = (TBOOL) FALSE;
    pScsiPhysDev->syncReserved2 = 0;
    pScsiPhysDev->syncReserved1 = 0;
    pScsiPhysDev->syncXferPeriod = 0;
    pScsiPhysDev->syncXferOffset = 0;
    /* Clear msg out request */
    pScsiPhysDev->useMsgout = SCSI_NO_MSGOUT;
    return((UINT *) &ackMsg1);
}

    /* if the previous msg was a unknown send a no op message */
    *pNcrCtl->pMsgOutLong = (UINT8)SCSI_MSG_NO_OP;
    pNcrCtl->msgOutCount = 1;
    return((UINT *) &ackAtnMsg);
}

/*******************************************************************************/
/* ncr810CheckStatRegs - Handle the ncr810 status registers*/
/* Check and process the copy of the SSTAT0 and DSTAT ncr810 registers.*/
/* Look through all register bits to find out the errors.*/
/* If an error implies to restart the script, it is returned to <*localStart>*/
/* and the return status is set to OK. Else returns ERROR.*/
/* RETURNS: OK, ERROR.*/
/* NOMANUAL*/
/* */
LOCAL STATUS ncr810CheckStatRegs
{
    SIOP *pSiop, /* pointer to scsiCtrl info */
    SCSI_PHYS_DEV *pScsiPhysDev, /* pointer to physDev info */
    UINT **localStart /* return script address */
}

    {UINT8 scratch_sist0;
    UINT8 scratch_sist1;
    UINT8 scratch;
    UINT tempReg;
    STATUS stat;
    UINT8 *pMsgOut;
    /* local pointer to msg out array */
    pMsgOut = SCSI_GET_PTR_MSGOUT(pScsiPhysDev);
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/* check scsi Error */
scratch_sist0 = (UINT8) pSiop->saveSist0; /* used to be stat0 */
scratch_sist1 = (UINT8) pSiop->saveSist1;
/* This is a strange a error - I got a function complete
* and a time out ???? That happened just after the
* select.(function complete = device selected).
*/
if ( (scratch_sist0 & (B_FCMP) ) && (scratch_sist1 & (B_STO)) )
{
    LOG_ERR("timeout during phase other than selection...continuing\n");
    *localStart = (UINT *) &asortPh;
    return(OK);
}
/* check for mismatch phase error. This case is handle at interrupt
* level. */
if ((scratch_sist0 & B_MA) == B_MA)
{
    LOG_ERR("mismatch at task level\n");
    LOG_ERR_CTX;
}
/* check for Function complete */
if ((scratch_sist0 & B_FCMP) == B_FCMP)
{
    LOG_ERR("scsi function complete intr\n");
    LOG_ERR_CTX;
}
/* Check for timeout error */
if ((scratch_sist1 & B_STO) == (UINT8) B_STO)
{
    if (pScsiPhysDev->devStatus == SELECT_REQUESTED)
    {
        pScsiPhysDev->devStatus = SELECT_TIMEOUT;
        errnoSet (S_scsiLib_SELECT_TIMEOUT);
    }
    LOG_ERR("ncr810Phase : scsi time out detected\n");
    LOG_ERR_CTX;
    /* Clear internal chip fifos */
    stat = NCR810_READ(pSiop->pCtest3,1,
        (void *)&tempReg); /* read Ctest3 */
}
(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

```c
    tempReg |= B_CLF;
    stat = NCR810_WRITE( pSiop->pCtest3,1,tempReg);

    /* check for selected interrupt */
    if ((scratch_sist0 & B_SEL) == B_SEL)
    {
        LOG_ERR("scsi selected intr\n");
        LOG_ERR_CTX;
    }

    /* check for scsi gross error */
    if ((scratch_sist0 & B_SGE) == B_SGE)
    {
        LOG_ERR("scsi gross error\n");
        LOG_ERR_CTX;
    }

    /* check for unexpected disconnect */
    if ((scratch_sist0 & B_UDC) == B_UDC)
    {
        /* Check if some rst device msg have been send */
        if (pMsgOut[0] == SCSI_MSG_BUS_DEVICE_RESET)
        {
            LOG_ERR("unexpected disconnect because target reseted\n");
            LOG_ERR_CTX;
            /* Clear sync capability in physDev structure */
            pScsiPhysDev->syncXfer = (TBOOL) FALSE;
            /* Clear Hardware sync register */
            stat = NCR810_WRITE( pSiop->pSbcl,1,0);
            stat = NCR810_READ(pSiop->pScntl1,1,
                               (void *)&tempReg); /* read Scntl1*/
            tempReg &= B_DHP; /* Why AND? */
            stat = NCR810_WRITE( pSiop->pScntl1,1,tempReg);  
        }
        else
        {
            LOG_ERR("scsi unexpected disconnect\n");
            LOG_ERR_CTX;
            /* Clear msg out request */
            pScsiPhysDev->useMsgout = SCSI_NO_MSGOUT;
        }
    }

(continued on next page)
```
/* check for reset received */
if ((scratch_sist0 & B_RST1) == B_RST1)
{
    LOG_ERR("scsi reset received\n");
    LOG_ERR_CTX;
}

/* check for parity error */
if ((scratch_sist0 & B_PAR) == B_PAR)
{
    LOG_ERR("scsi or bus parity error\n");
    LOG_ERR_CTX;
}

/* check Dma error */
scratch = pSiop->saveDstat;

/* check for illegal instruction */
if (scratch & B_IID)
{
    LOG_ERR("ncr810 illegal instruction\n");
    LOG_ERR_CTX;
}

/* check for bus fault */
if (scratch & B_BF)
{
    LOG_ERR("ncr810 access memory error\n");
    LOG_ERR_CTX;
}

/* check for abort condition */
if (scratch & B_ABT)
{
    LOG_ERR("ncr810 abort bit set\n");
    LOG_ERR_CTX;
}

/* all of these conditions are fatal */
return(ERROR);

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

/*******************************************************************************
* ncr810FlushCache - Flush cache entry.
* It flushes all memory space used by the chip to transfer data from memory to
* scsi.
* All of the data space to perform a transaction must be set before calling it.
* It uses at interrupt level to start a new scsi command with a safe cache
* context.
* RETURN: nothing.
* NOMANUAL
*/
LOCAL void ncr810FlushCache
    (...)
    { /* Flush user data buffer if it's a scsi write */
        if (pScsiPhysDev->pScsiXaction->dataDirection == O_WRONLY)
            CACHE_USER_FLUSH (pNcrCtl->pDataLong, pNcrCtl->dataCount);
        /* Flush scsi command buffer */
        CACHE_USER_FLUSH (pNcrCtl->pCmdLong, pNcrCtl->cmdCount);
        /* Flush msgout array with the maximum msgout length possible */
        CACHE_USER_FLUSH (pNcrCtl->pMsgOutLong, MAX_MSG_OUT_BYTES + 1);
    }
(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOP SCSI Driver Listing

*******************************************************************************
*
* ncr810StepEnable - Enable/disable script single step
* Enable/disable single step facility on ncr chip. Also, unmask/mask single
* step interrupt in Dien register. To use the single step feature you must
* disable all the script code that turns on the SCSI timeout.
* This instruction is used in the script to enable the time out only before
* a select phase and awaits DISCONNECT.
*
* To disable it just put a ";" to the beginning of the line and recompile
* the driver. Before any SCSI execution you just have to use
* ncr810StepEnable (pSiop, TRUE) to run in single step mode.
* To run a script step use ncr810SingleStep(pSiop).
* To disable use ncr810StepEnable (pSiop, FALSE).
*
* RETURNS: N/A
*
* NOTE: In the future it might be good to use a
* scratch register on the chip for use by the script
* enable/disable the Timeout feature at start of
* the run time of the script.
*
* NOMANUAL
*/

VOID ncr810StepEnable
(
    SIOP *pSiop,  /* pointer to SIOP info */
    TBOOL boolValue  /* TRUE/FALSE to enable/disable single step */
)
{
    UINT tempReg;
    STATUS stat;
    if (boolValue == TRUE)
    {
        /* Disable intr timeout */
        stat = NCR810_READ(pSiop->pSien1,1,
            (void *)&tempReg); /* read Sien1 */
        tempReg &= ~B_STO;
        stat = NCR810_WRITE( pSiop->pSien1,1,tempReg);
        /* Set global state */
        siopSStep = TRUE;
    }

} (continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOF SCSI Driver Listing

    stat = NCR810_READ(pSiop->pDien,1, 
        (void *)&tempReg); /* read dien */
    tempReg |= B_SSI; /* Single step */
    stat = NCR810_WRITE( pSiop->pDien,1,tempReg); 
    /* enable single step interrupt */
    stat = NCR810_READ(pSiop->pDcntl,1, 
        (void *)&tempReg); /* read Dcntl*/
    tempReg |= B_SSM; /* Sing. step mode */
    stat = NCR810_WRITE( pSiop->pDcntl,1,tempReg); 
    /* set single step mode */
    stat = NCR810_READ(pSiop->pDmode,1, 
        (void *)&tempReg); /* read Dmode*/
    tempReg &= ~B_MAN; /* Disable man start */
    stat = NCR810_WRITE( pSiop->pDmode,1,tempReg); 
    /* disable manual Start */
    /* Disable time out on the chip */
    /* Init Single Step semaphore */
    semBInit (&pSiop->singleStepSem, SEM_Q_FIFO, SEM_EMPTY);
}

else
{
    stat = NCR810_READ(pSiop->pSien1,1, 
        (void *)&tempReg); /* read sien */
    tempReg |= B_STO; /* enable interrupt timeout */
    stat = NCR810_WRITE( pSiop->pSien1,1,tempReg); 
    /* Enable intr timeout */
    /* Set global state */
    siopSStep = FALSE;
    stat = NCR810_READ(pSiop->pDcntl,1, 
        (void *)&tempReg); /* read Dcntl */
    tempReg &= ~B_SSM; /* unset sing. step */
    stat = NCR810_WRITE( pSiop->pDcntl,1,tempReg); 
    /* unset single step mode */
    (continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIOP SCSI Driver Listing

    stat = NCR810_READ(pSiop->pDien,1, (void *)&tempReg); /* read Dien */
    tempReg &= ~B_SSI; /* Disable script int. */
    stat = NCR810_WRITE( pSiop->pDien,1,tempReg); /* disable script interrupt */
    stat = NCR810_READ(pSiop->pDmode,1, (void *)&tempReg); /* read Dmode */
    tempReg |= B_MAN; /* Disable script int. */
    stat = NCR810_WRITE( pSiop->pDmode,1,tempReg); /* enable manual Start */
}

/*******************************************************************************
* ncr810SingleStep - Perform a single step
* Perform a single step by writing the STD bit in the DCNTRL register.
* The parameter is a pointer to the SIOP information.
* RETURNS: N/A
* NOMANUAL
*/

VOID ncr810SingleStep
{
    SIOP * pSiop /* pointer to SIOP info */
}
{
    UINT *pAdd;
    STATUS stat;
    UINT tempReg;
    stat = NCR810_READ(pSiop->pDcntl,1, (void *)&tempReg); /* read Dcntl */
    tempReg |= B_STD; /* Disable script int. */
    stat = NCR810_WRITE( pSiop->pDcntl,1,tempReg);
    semTake (&pSiop->singleStepSem, WAIT_FOREVER);

(continued on next page)
Example 8–6 (Cont.) NCR 53C810 PSIOp SCSI Driver Listing

```c
stat = NCR810_READ(pSiop->pDsp,4, 
    (void *)&pAdd); /* read Dsp */

pAdd--; 
pAdd--; 

printf("ncr810 Cur Opcode @0x%08x: 0x%08x ", (int) pAdd, *pAdd); pAdd++;
printf("0x%08x\n",*pAdd); pAdd++;
printf("ncr810 Next Opcode @0x%08x: 0x%08x ", (int) pAdd, *pAdd); pAdd++;
printf("0x%08x\n",*pAdd);
ncr810Show ((SCSI_CTRL *) pSiop);
}

/******************************************************************************
 * ncr810SetIntrTimeout - Enable / disable SCSI interrupt timeout 
 * 
 * Enable or disable SCSI timeout interrupt. 
 * The parameter is a pointer to the SIOP information. 
 * 
 * RETURNS: N/A 
 * 
 * NOMANUAL 
 */
void ncr810SetIntrTimeout
{
    SIOP * pSiop, /* pointer to SIOP info */
    TBOOL boolVal /* TRUE enable time out */
}

UINT tempReg;
STATUS stat;

stat = NCR810_READ(pSiop->pSien1,1, 
    (void *)&tempReg); /* read Sien1 */

if (boolVal == TRUE)
    tempReg |= B_STO; /* Disable man start */
else
    tempReg &= ~B_STO; /* Disable man start */

stat = NCR810_WRITE(pSiop->pSien1,1,tempReg);
}

(continued on next page)
```
Example 8–6 (Cont.) NCR 53C810 PSIO SCSI Driver Listing

/*******************************************************************************
* ncr810SelDevTimeout - do nothing
* This entry point is here to satisfy scsilib. This option is not supported
* with ncr810 chip. The only SCSI time out value is 250ms.
* RETURNS: N/A
* NOMANUAL
*/

LOCAL void ncr810SelTimeOutCvt(
    FAST SCSI_CTRL *pScsiCtrl, /* ptr to SCSI controller info */
    FAST UINT timeOutInUsec, /* timeout in microsecs */
    FAST UINT *pTimeOutSetting /* ptr to result */
){
}

#ifdef NCR810_INT_LOG_MEM
/*******************************************************************************
* ncr810LogShow - Show a log memory interrupt.
* Show a log memory result in a friendly way. To use only in trace debug way.
* This routine uses a global pointer pLogMemShow to display a log mem node.
* NOMANUAL
*/

(continued on next page)
Example 8–6 (Cont.)  NCR 53C810 PSIO P SCSI Driver Listing

void ncr810LogShow
{
    printf("counIntr: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("logCount: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("dps: 0x%08x\n",*pLogMemShow);pLogMemShow++;
    printf("pNcrCtl: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("pNcrCtlCmd:0x%08x",*pLogMemShow);pLogMemShow++;
    printf("pScsiPhysDev:0x%08x",*pLogMemShow);pLogMemShow++;
    printf("pPhysDevNcr:0x%08x",*pLogMemShow);pLogMemShow++;
    printf("pData: 0x%08x\n",*pLogMemShow);pLogMemShow++;
    printf("dataCount: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("cmdRequest: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("sbcl: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("sdid: 0x%08x",*pLogMemShow);pLogMemShow++;
    printf("istat: 0x%08x\n",*pLogMemShow);pLogMemShow++;
    pLogMemShow++;
}
#endif /* NCR810_INT_LOG_MEM */
Example 8–7  NCR 53C810 PSIOP SCSI Scripts Module Listing

/* ncr810init.c - script programs for the NCR 810 */

/* Copyright 1989-1992 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*

INTERNAL
NCR810 FEATURE USED
This chip has code which performs some action on the scsi bus.
The description level of the code is close to an assembly language and
dedicated for scsi bus operations.
The opcode is a pair of 32bit word that allow some basic operations in the
code (jump, tests values) and on the scsi (select ,assert line):

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 SBFR
 ........
transfert control instructions
 jump <Label>
 int <value> when PHASE_NAME
 ........
 move memory instruction
 never use.

The previous example does not included all of the instruction set,
see the NCR data reference manual.
Another key point is the capability of the script to perform some
limited indirect addressing to get information from the cpu memory.
This is used to get Target ID to select it, to get data, message pointer
and count during transfer phase. This avoids having to patch the script
at the run time to specify count, data pointer and target ID.

Script can transfer directly, with its DMA core, a data block from/to the
scsi bus from/to cpu memory.
The instruction involved is:
move from <offset> when PHASE_NAME

In this case the offset is hard coded in the ncr810Script.h and indexed the
right field in the NCR_CTL structure (see ncr810Script.h).

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

The field looks like:
  aligned 32 bit @->00xxxxxx 24bit count
  ->ZZZZZZZZ 32bit address pointer.

The offset is added to the content of the DNAD register, loaded at the run time with the base address of the structure.

In the same way the select instruction uses a field like:
  select from <offset>,<Alternate jump @>

where offset + DNAD points to a field like:
aligned 32 bit@->0000|0000|xxxx xxxx|xPPP |SSSS|0000|0000
Target ID|Period|Offset

The Period and offset are used when a target supports the synchronous transfer.

INTERRUPT SOURCE

The chip has three main kind of interrupt, scsi, dma interrupt and script interrupt. The scsi interrupts are maskable individually with the sien register.

.CS
SIEN
  7  M/A Mismatch interrupt(initiator)/ ATN assert (Target)
     Used in the driver to detect a phase change during data/msg transfer. The ncr is never used in target mode.
  6  FCMP function complete Not used (masked).
  5  STO scsi Bus Time out
     Used to get timeout on select.(**)
  4  SEL selected or reselected Not used in the driver.(masked).
  3  SGE scsi Gross error. Used in the driver.
  2  UDC unexpected disconnect.
     Used to detect unexpected target disconnection or an expected disconnect if the target received a "device reset message".
  1  RST scsi reset received (not used).(masked)(***)
  0  PAR parity error not used.(masked)(***)

(**) The script disable the timeout capability by setting the bit 7 in CTEST7 register. Because the ncr will generate also a timeout interrupt if no activity occur on the scsi bus after 250ms, and not only during a scsi select.

(*** This case should be processed, but not included in the driver today.

.CE

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

The Dma interrupts are maskable with the DIEN register:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIEN</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>BF bus fault. Memory access error.</td>
</tr>
<tr>
<td>4</td>
<td>Aborted, means that the abort bit have been set in ISTAT reg. This case never happen (Abort bit not used).</td>
</tr>
<tr>
<td>3</td>
<td>SSI single step interrupt. Not used in the operational mode. This bit is used in single step debug mode in the driver, allowing to execute script step by step.</td>
</tr>
<tr>
<td>2</td>
<td>SIR script interrupt, used to detect a script instruction &lt;int ...&gt;.</td>
</tr>
<tr>
<td>1</td>
<td>WTD Watchdog interrupt. This case never happens because the watchdog timeout value is set to 0 in DWT register.</td>
</tr>
<tr>
<td>0</td>
<td>IID illegal instruction detected. Received in two cases: 1-Bad opcode, 2-count set to 0 in the &lt;move&gt; opcode.</td>
</tr>
</tbody>
</table>

The other register involved in the interrupt process is ISTAT. This register is the ONLY READABLE/WRITABLE register during a script execution.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISTAT</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ABRT Abort current instruction. Not used</td>
</tr>
<tr>
<td>6</td>
<td>RST software reset used to set the chip to a known state.</td>
</tr>
<tr>
<td>5</td>
<td>SIGP Signal process bit. This bit is used in the ncr driver to support the connect disconnect to get an interrupt to start a new scsi command. (SEE CONNECT/DISCONNECT ...).</td>
</tr>
<tr>
<td>4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3</td>
<td>Connected, not used.</td>
</tr>
<tr>
<td>2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>1</td>
<td>SIP scsi interrupt pending. Used at interrupt level to detect and clear SIEN interrupt.</td>
</tr>
<tr>
<td>0</td>
<td>DIP dma interrupt pending. Used at interrupt level to detect and clear DIEN interrupt.</td>
</tr>
</tbody>
</table>

The interrupt are cleared by reading the register source, and by checking the DIP and SIP bit in ISTAT. Because the chip has a stack interrupt, the DIEN and SIEN registers are read until the ISTAT bits are cleared to be sure there is no interrupt pending.

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

The last interrupt source is the script <INT> opcode. This instruction is used to interrupt the CPU if the SCSI transaction involves a CPU process.

```
int <int value> [<condition><value>] (pg 6-10 of the programmers guide).
```

The <int value> is used in the driver code to detect the SCSI condition and takes a decision for the next step of the transaction. This value is loaded from the opcode and read by the CPU in the DSPS register.

CONNECT/DISCONNECT AND NEW COMMAND START

When connect/disconnect is enabled, the target could reconnect at any time. This is notified to the target by an Identify message sent by the initiator after a successful selection of the target. The disconnect is always sent to an initiator by a DISCONNECT message. The reconnect could occur at any arbitration phase. To be able to detect it at any time the chip has to wait for a reconnect on the SCSI, which is done with the <wait reconnect> script opcode:

```
wait RECONNECT,<Alternate @><next opcode>
```

If a reconnect occurs, then the <next opcode> is executed. If the SIGP bit is set the script jump to <Alternate @>.

```
CE
```

But a reconnect could also occur during a <select> opcode for another device (it looses the arbitration). The <wait RECONNECT> instruction is the idle script entry point. They are two ways to abort this instruction:

1-The SIGP bit is set
2-A reconnect, or a select occur, in this case select means that the initiator is selected as a target on the SCSI bus

The SIGP bit is used to detect and start a new command at interrupt level. It is set at task level each time that a new command has to be run.

```
SCRIPT START FLOW CHART
```

The reselected path is driven by a physical SCSI reconnect/select.

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

IDLE ENTRY POINT

<waitSelect>
Wait for a reselect, Alternate @-------->
and test sigp ? <-------------------->

| SIGP set or Select to start |
|<reselect> |
| / No <checkNewCmd> |

|<reselect> |
|--------Check if we are connected ? |
|<reselect> |
| clear SIGP |
| save Targ/init ID |
| Wait for reselect, @select as a target |
| from LCRC register |
| Jump to <reselect> |
| Get identify |
|<startNewCmd> |
| test and clear SIGP |
| int Cpu <SELECT AS A TARGET> |
| 1) SIGP set |
| int Cpu <NEW COMMAND> |
| 2) SIGP clear |
| int Cpu <BAD NEW CMD> |

.CE

a) RESELECT
At interrupt level, the target ID from the saved LCRC register (LCRC contain
the data bus value that is <the target ID> | <initiator ID>) and the LUN
extracted from the identify message sent by the target are used to index
an array that keep track of each PHYS_DEV nexus (Max = 64) in the SIOP
structure (driver structure).

b) NEW COMMAND
A global variable <pNcrCtlCmd> in the SIOP structure allows the script to be
start at a selected entry point with the script address included in this
current nexus pointer in pNcrCtlCmd.

c) SELECT AS A TARGET
That an error case (we dont support the target mode) , restart script at
the IDLE point.

d) BAD NEW CMD
It’s a bogus start command restart script at the IDLE Point.

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

SCRIPT REMARKS
This script performs a scsi transaction. This script is a part of the usual scsi phase routine. This code is only compatible with the ncr810 because it uses an indirect addressing mode to avoid relocation at run time.
All of the external values will be relative to the DSA register.
DSA register has to be loaded with the address of the ncrCtrl host memory structure (see ncr810Script.h).
Scratch3 register is used to save the LCRC register. LCRC is stamped with the initiator ID and target ID when a reconnect (initiator is selected by a target) occurs.
SCRATCH0 is used to hold phase requested to be able to process a phase mismatch during any data or message phase. Usually that occur before a legal disconnect from the target (save pointer and disconnect message).

#define NCR_COMPILER_BUG
#include"drv/scsi/ncr810Script.h"

extern UINT waitSelect[];
extern UINT reselect[];
extern UINT checkNewCmd[];
extern UINT startNewCmd[];
extern UINT selWithAttn[];
extern UINT selWithoutAttn[];
extern UINT contTransact[];
extern UINT endTransact[];
extern UINT msgOut1[];
extern UINT msgOut3[];
extern UINT outputData[];
extern UINT checkOut[];
extern UINT inputData[];
extern UINT msg1[];
extern UINT msg2[];
extern UINT msg3[];
extern UINT selectAddr[];
extern UINT ackMsg1[];
extern UINT ackMsg3[];
extern UINT checkPhData[];
extern UINT ackAttnMsg[];
extern UINT asortPh[];
extern UINT endAbort[];
extern UINT reserOutPh[];
extern UINT reserInPh[];
extern UINT reselSelect[];
extern UINT last_datap[];

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
UINT last_datap_idx EQ 0;
UINT waitSelect_idx EQ 1;
UINT reselect_idx EQ 2;
UINT checkNewCmd_idx EQ 3;
UINT startNewCmd_idx EQ 4;
UINT reselSelect_idx EQ 5;
UINT selWithAttn_idx EQ 6;
UINT selWithoutAttn_idx EQ 7;
UINT contTransact_idx EQ 8;
UINT endTransact_idx EQ 9;
UINT msgOut1_idx EQ 10;
UINT msgOut3_idx EQ 11;
UINT outputData_idx EQ 12;
UINT checkOut_idx EQ 13;
UINT inputData_idx EQ 14;
UINT msg1_idx EQ 15;
UINT msg2_idx EQ 16;
UINT msg3_idx EQ 17;
UINT selectAddr_idx EQ 18;
UINT ackMsg1_idx EQ 19;
UINT ackMsg3_idx EQ 20;
UINT checkPhData_idx EQ 21;
UINT ackAttnMsg_idx EQ 22;
UINT asortPh_idx EQ 23;
UINT reserOutPh_idx EQ 24;
UINT reserInPh_idx EQ 25;
UINT endAbort_idx EQ 26;
UINT script_ptr_sz EQ 108;

UINT last_datap[] = {
    0x98080000, 0x00000000
};

UINT waitSelect[] = {
    0x7C41FB00, 0x00000000, /* move sien1&~B_STO to sien1 - disable timeouts */
    0x78340A00, 0x00000000, /* move PHASE_NOPHASE to SCRATCH0 */
    0x50000000, 0xFEEDFACE, /* wait RESELECT PASS ((UINT)checkNewCmd) */
    0x80080000, 0xFEEDFACE /* jump PASS ((UINT)reselect) */
};

UINT reselect[] = {
    0x60000200, 0x00000000,
    0x6A370000, 0x00000000,
    0x1F000000, 0x34,
    0x98080000, RECONNECT_PROCESS
};

(continued on next page)
```

8–140 Writing a SCSI Device Driver
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
UINT checkNewCmd[] = {
  0x74011000, 0x00000000,
  0x800C0000, 0xFEEDFACE,  
  0x741A2000, 0x00000000,  
  0x50000000, 0xFEEDFACE,
  0x80080000, 0xFEEDFACE
};

UINT startNewCmd[] = {
  0x741A2000, 0x00000000,
  0x80840000, 0x00000010,  
  0x98080000, BAD_NEW_CMD,  
  0x80080000, 0xFEEDFACE,  
  0x98080000, NEW_COMMAND_PROCESS
};

UINT reselSelect[] = {
  0x98080000, RECONNECT_IN_SELECT
};

UINT selWithAtn[] = {
  0x7A410400, 0x00000000, /* move sien1|b_timeo to sien1 - allow timeouts */  
  0x78340A00, 0x00000000,  
  0x43000000, 0xFEEDFACE,  
  0x830B0000, 0xFEEDFACE,  
  0x860B0000, 0xFEEDFACE,  
  0x9E020000, MSGOUT_EXPECT
};

UINT selWithoutAtn[] = {
  0x7A410400, 0x00000000, /* move sien1|b_timeo to sien1 - allow timeouts */  
  0x78340A00, 0x00000000,  
  0x42000000, 0xFEEDFACE,  
  0x80080000, 0xFEEDFACE /* gets here ok */
};
```

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```
UINT contTransact[] = {
0x830B0000, 0xFEEDFACE,
0x7C41FB00, 0x00000000, /* move sien1&~B_STO to sien1 - new location!!! */
0x860B0000, 0xFEEDFACE,
0x830B0000, 0xFEEDFACE,
0x870B0000, 0xFEEDFACE,
0x9A020000, BAD_PH_BEFORE_CMD,
0x78340200, 0x00000000,
0x1A000002, 0x1c,
0x870B0000, 0xFEEDFACE,
0x830A0000, 0xFEEDFACE,
0x810A0000, 0xFEEDFACE,
0x800A0000, 0xFEEDFACE,
0x98080000, BAD_PH_AFTER_CMD
};

UINT endTransact[] = {
0x7C41FB00, 0x00000000, /* move sien1&~B_STO to sien1 - disallow */
0x78340300, 0x00000000,
0x1B000003, 0x2C,
0x9F030000, MSGIN_EXPECT_AFTER_ST,
0x78340700, 0x00000000,
0x1F000007, 0xC,
0x98040000, BAD_MSG_INSTEAD_CMDCOMP,
0x60000040, 0x00000000, /* clr ack */
0x48000004, 0x00000000, /* move 0 to scntl0 - clears sdu bit */
0x98080000, GOOD_END
};

UINT msgOut1[] = {
0x7C41FB00, 0x00000000, /* move sien1&~B_STO to sien1 - disallow */
0x78340300, 0x00000000,
0x1E000004, 0x04,
0x820B0000, 0xFEEDFACE,
0x830A0000, 0xFEEDFACE,
0x810A0000, 0xFEEDFACE,
0x800A0000, 0xFEEDFACE,
0x98080000, PH_UNKNOWN
};
```
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
UINT msgOut3[] = {
0x78340600, 0x00000000, 0x1E000006, 0x04, 0x820B0000, 0xFEEDFACE,
0x830A0000, 0xFEEDFACE, 0x810A0000, 0xFEEDFACE, 0x800A0000, 0xFEEDFACE,
0x870A0000, 0xFEEDFACE, 0x98080000, PH_UNKNOWN
};
UINT outputData[] = {
0x78340000, 0x00000000, 0x18000000, 0x24,
};
UINT checkOut[] = {
0x830B0000, 0xFEEDFACE,
0x870B0000, 0xFEEDFACE, 0x80080000, 0xFEEDFACE
};
UINT inputData[] = {
0x78340100, 0x00000000, 0x19000001, 0x24, 0x80080000, 0xFEEDFACE
};
UINT msg1[] = {
0x78340700, 0x00000000, 0x1F000007, 0x0C, 0x808C0001, 0x00000028, 0x980C0002, SAVDATP_BEFORE_CMD, 0x980C0003, RESTORE_POINTER,
0x808C0004, 0x00000028, 0x980C0007, REJECT_MSG1, 0x98080000, BAD_MSGIN BEFORE_CMD,
0x78020000, 0x00000000, /* move 0 to scntl0 - clears sdu bit */
0x60000040, 0x00000000,
0x1F000007, 0x14, /* move ext. msg */
0x98080000, EXTMSG BEFORE_CMD, /* interrupt */
};
```
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
0x78020000, 0x00000000, /* move 0 to scntl0 - clears sdu bit */
0x60000040, 0x00000000,
0x48000000, 0x00000000,
0x98080000, DISC_BEFORE_CMD
```

```c
UINT msg2[] = {
    0x78340700, 0x00000000,
    0x1F000007, 0x0C, 0x808C0001, 0x00000020, 0x980C0002, SAVDATP_AFTER_CMD,
    0x980C0003, RESTORE_POINTER,
    0x98080000, DISC_AFTER_CMD
};
```

```c
UINT msg3[] = {
    0x78340700, 0x00000000,
    0x1F000007, 0x0C, 0x808C0001, 0x00000020, 0x980C0002, SAVDATP_AFTER_DATA,
    0x980C0003, RESTORE_POINTER,
    0x98080000, DISC_AFTER_DATA
};
```

```c
UINT selectAddr[] = {
    0x98080000, SELECT_AS_TARGET
};
```

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIO SCSI Scripts Module Listing

```c
UINT ackMsg1[] = {
    0x870B0000, 0xFEEDFACE,
    0x80080000, 0xFEEDFACE
};

UINT ackMsg3[] = {
    0x78020000, 0x00000000, /* move 0 to scntl0 - clears sdu bit */
    0x60000040, 0x00000000,
    0x870B0000, 0xFEEDFACE,
    0x80080000, 0xFEEDFACE
};

UINT checkPhData[] = {
    0x810B0000, 0xFEEDFACE,
    0x800A0000, 0xFEEDFACE,
    0x860A0000, 0xFEEDFACE,
    0x830A0000, 0xFEEDFACE,
    0x820A0000, 0xFEEDFACE,
    0x870A0000, 0xFEEDFACE,
    0x840A0000, 0xFEEDFACE,
    0x850A0000, 0xFEEDFACE,
    0x98080000, PH_UNKNOWN
};

UINT ackAtnMsg[] = {
    0x58000008, 0x00000000,
    0x78020000, 0x00000000, /* move 0 to scntl0 - clears sdu bit */
    0x60000040, 0x00000000,
    0x78340600, 0x00000000,
    0x1E000006, 0x04,
    0x86030000, 0xFEEDFACE,
    0x98080000, PH_UNKNOWN
};

UINT asortPh[] = {
    0x60000008, 0x00000000,
    0x810B0000, 0xFEEDFACE,
    0x800A0000, 0xFEEDFACE,
    0x860A0000, 0xFEEDFACE,
    0x830A0000, 0xFEEDFACE,
    0x820A0000, 0xFEEDFACE,
    0x870A0000, 0xFEEDFACE,
    0x840A0000, 0xFEEDFACE,
    0x850A0000, 0xFEEDFACE,
    0x98080000, PH_UNKNOWN
};
```

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
};
UINT reserOutPh[] = {
    0x78340400, 0x00000000,
    0x98080000, RES_OUT_DETECTED
};
UINT reserInPh[] = {
    0x78340500, 0x00000000,
    0x98080000, RES_IN_DETECTED
};
UINT endAbort[] = {
    0x58000008, 0x00000000,
    0x60000040, 0x00000000,
    0x78340600, 0x00000000,
    0x1E000006, 0x04,
    0x48000000, 0x00000000,
    0x98080000, ABORT_CLEAR_END
};
```

(continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

```c
UINT relocation[] = {
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(last_datap),
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(waitSelect),
    0x80080000, 0x00000014,
    0x80080000, 0x0000001C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(reselect),
    0x80080000, 0x0000001C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(checkNewCmd),
    0x80080000, 0x0000000C,
    0x80080000, 0x0000001C,
    0x80080000, 0x00000024,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(startNewCmd),
    0x80080000, 0x0000000C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(reselSelect),
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(selWithAtn),
    0x80080000, 0x00000014,
    0x80080000, 0x0000001C,
    0x80080000, 0x0000001C,
    0x80080000, 0x00000024,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(selWithoutAtn),
    0x80080000, 0x00000014,
    0x80080000, 0x0000001C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(contTransact),
    0x80080000, 0x00000004,
    0x80080000, 0x0000000C,
    0x80080000, 0x00000014,
    0x80080000, 0x0000001C,
    0x80080000, 0x00000003C,
    0x80080000, 0x000000044,
    0x80080000, 0x00000004C,
    0x80080000, 0x000000054,
    0x80080000, 0x00000005C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(endTransact),
    0x80080000, 0x00000004,
    0x80080000, 0x00000002C,
    0x98080000, 0xFEEDFACE,
    0x0F000001, sizeof(msgOut1),

    // (continued on next page)
Example 8–7 (Cont.) NCR 53C810 PSIOP SCSI Scripts Module Listing

0x80080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x98080000, 0x0000000C,
0x98080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
Example 8–7 (Cont.) NCR 53C810 PSIOOP SCSI Scripts Module Listing

```c
0x80080000, 0x00000014,
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(checkPhData),
0x80080000, 0x00000004,
0x80080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x80080000, 0x0000003C,
0x80080000, 0x00000044,
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(ackAtnMsg),
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(asortPh),
0x80080000, 0x0000000C,
0x80080000, 0x00000014,
0x80080000, 0x0000001C,
0x80080000, 0x00000024,
0x80080000, 0x0000002C,
0x80080000, 0x00000034,
0x80080000, 0x0000003C,
0x80080000, 0x00000044,
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(reserOutPh),
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(reserInPh),
0x98080000, 0xFEEDFACE,
0x0F000001, sizeof(endAbort),
0x80080000, 0x0000001C,
0x78020000, 0x00000000, /* move 0 to scnt10 - clears sdu bit */
0x60000040, 0x00000000
```

Writing a SCSI Device Driver  8–149
Part III
Porting Existing VxWorks Device Drivers to the Alpha Platform

Part III describes how to port existing VxWorks device drivers to an Alpha-based platform. While the emphasis here is on drivers written for Motorola 68K targets, the information applies equally to drivers written for other 32-bit VxWorks platforms.

This part consists of the following chapters:

• Chapter 9, Porting Considerations
• Chapter 10, Using the 68K VMEbus Library Interface
• Chapter 11, Driver Porting Example
Porting Considerations

The 64-bit environment of the VxWorks Alpha targets differs from that of other VxWorks targets in a number of ways that affect your programs.

• The 64-bit Alpha architecture forces changes in the way data is represented and aligned.
• The Alpha architecture permits a minimum data granularity of 32 bits.
• As a true RISC architecture, the Alpha architecture does not provide certain processor instructions typically found under a CISC architecture.

This chapter describes these differences and the changes you might have to make to your existing 32-bit VxWorks device drivers. It covers the following topics:
• Data representation, Section 9.1
• Data access, Section 9.2
• Data alignment, Section 9.3
• Constants, Section 9.4
• Variables, Section 9.5
• Structures, Section 9.6
• Pointers, Section 9.7
• File systems, Section 9.8
• Unavailable instructions, Section 9.9
• The lint command, Section 9.10
9.1 Data Representation

Under VxWorks for Alpha, the C language includes new 64-bit data types. Table 9–1 compares data type lengths between 32-bit platforms and the 64-bit Alpha platforms.

Table 9–1  C Language Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>32-Bit Platform Size in Bits</th>
<th>64-Bit Alpha Platform Size in Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>long long</td>
<td>Not available</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>32 (IEEE single precision)</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64 (IEEE double precision)</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>32 (64)</td>
<td>64</td>
</tr>
</tbody>
</table>

As the table shows, the major differences are that on Alpha platforms: long is defined to be 64 bits; pointer is defined to be 64 bits, extending the address space; and long long is a new data type, defined to be 64 bits. The long long type provides a unique name for a 64-bit data type, providing a way to specifically identify 64-bit quantities.

Note that Alpha platforms use right-to-left byte ordering (little endian) for integer types.

The most common source of difficulty when porting programs from a 32-bit platform to a 64-bit Alpha platform comes from improperly sized pointers. On 32-bit platforms, pointer and int are equivalent (32 bits). An Alpha platform, on the other hand, uses 64-bit pointers.

Section 9.7 describes this problem in more detail.

9.2 Data Access

Unlike platforms that allow hardware access of byte and word values, the Alpha architecture supports hardware access of only longword (32 bits) or quadword (64 bits) values. Byte and word accesses must be accomplished in the software using multiple instructions that load, mask, and shift longword or quadword values. (See the Alpha Architecture Reference Manual for explicit detail.)
In addition to increasing execution time, be aware that this software manipulation for byte and word access can produce incorrect results when your program accesses adjacent byte or words in a shared memory segment. If you are porting a multithreaded application or multiple processes that share memory, your programs must use some form of locking function to avoid conflicts when accessing adjacent bytes or words.

Also, the order in which write operations occur can differ from what you intended. If it is important to guarantee the order in which data is written to memory, you should use memory barrier instructions. (See the `sysWbFlush()` manual page for your target platform.)

### 9.3 Data Alignment

On Alpha platforms, as on other platforms, data alignment is implied by the data type. The Alpha architecture, however, supports a minimum data granularity of 32 bits and only permits hardware access of data that is “naturally” aligned on 32- and 64-bit boundaries.

VxWorks for Alpha contains software to handle unaligned data accesses, if necessary. However, while this handler is transparent to your programs, it will severely impact performance. To get the best performance, you should examine and redefine your structures so that elements are aligned on 32- or 64-bit boundaries, even if it means you must explicitly fragment a structure.

Section 9.6 describes this issue in more detail.

### 9.4 Constants

Some constants can have different values between 32- and 64-bit platforms, potentially changing the behavior of some operators and functions.

The following table lists some constants and their 32- and 64-bit values:

<table>
<thead>
<tr>
<th>C Constant</th>
<th>Value</th>
<th>Value (32-bit)</th>
<th>Value (64-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
<td>((2^{32}) - 1)</td>
<td>-1</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>4294967296</td>
<td>(2^{32})</td>
<td>1</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>0x100000000</td>
<td>(2^{32})</td>
<td>1</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>0xFFFFFFFFFFFFFFFF</td>
<td>((2^{64}) - 1)</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

\(^1\)Constant value cannot be represented in 32 bits
9.4.1 Assignment and Argument Passing

The way you specify a constant can affect its value when it is passed as a function argument. For example, suppose you want to specify an integer constant as 543210. In C, to specify a long int constant, you use the letter suffix L or l. To specify an unsigned long, you use the UL or ul suffix. (Note that uppercase L is preferred since lowercase l is easily confused with the number one.) Now note what happens when three differently specified constants are passed to a function:

```c
labs(543210);
labs(543210L);
labs(543210UL);
```

On an Alpha platform, 543210 is treated as a 4-byte constant. 543210L and 543210UL are treated as 8-byte constants. If the `labs()` function expects a long argument, each of these invocations will work as expected since the int constants are converted to long.

If, on the other hand, the `labs()` function expects type int, the long constant is truncated to an integer constant. This truncation will result in the loss of significant digits in the following circumstances:

- When a signed constant is greater than the maximum integer constant (INT_MAX) of +2147483647
- When a signed constant is less than the minimum integer constant (INT_MIN) of -2147483648
- When an unsigned constant is greater than the maximum unsigned integer constant (UINT_MAX) of 4294967295

Truncation can also be a problem in an assignment expression where a long integer constant is assigned to a variable of type int.

In cases like this, you should explicitly use the L or UL suffix. Also make sure that your constants are specified consistently with function arguments or variables to which they are assigned.

When you must pass zero to a pointer argument and no function prototype is visible, always use a value of NULL (defined in the stdio.h file). Using a value of 0 will result in a 4-byte zero instead of an 8-byte zero (0L). In a comparison, an assignment, or a function call where the correct function prototype is in scope, standard C promotion rules will be in effect and the correct value will be assigned.

To minimize assignment and argument errors, use function prototypes so the compiler can check the number and type of arguments when you call the functions.
9.4.2 Shift Operations

A bit shift operation on an integer constant yields a 32-bit constant. The following example results in value being assigned a 32-bit constant:

```c
long value;
value = 10 << 2;
```

The top 32 bits of value depend on the type of the value shifted. Signed values are sign extended; unsigned values are zero extended. If you need a result of type long, you must use the L or UL suffix for long integer constants.

9.5 Variables

The 64-bit Alpha environment changes assumptions about how you declare your variables and how you use them in assignments and as function arguments.

9.5.1 Declarations

Remember that a declaration that has no type specifier defaults to a type of int. For example, the following six declarations declare each variable to be of type int:

```c
extern e;
register n;
static x;
unsigned i;
const c;
function ();
```

9.5.2 Assignments and Function Arguments

Since pointer, int, and long are not the same size in the 64-bit Alpha environment, problems can arise depending on how the variables are assigned and used in your program. Use the following guidelines:

- Do not use int and long interchangeably since significant digits can be truncated. For example, avoid assignments like the following:

```c
int i;
long l;
i = l;
```

Use the lint \-Q command to help you find these potential problems. See Section 9.10 for more information about the lint command.
• Do not pass long arguments to functions expecting type int. For example, avoid assignments like the following:

```c
int toascii(int);
int i;
long l;
i = toascii(l);
```

• Do not exchange variables of type pointer and int. Assigning a pointer to an int, assigning it back to a pointer, and then dereferencing the pointer will produce incorrect results. For example, avoid assignments like the following:

```c
int i;
char *buffer;
buffer = (char *)malloc(MAX_LINE);
i = (int)buffer;
buffer = (char*)i;
```

Use the lint \(-Q\) command to find these “pointer-to-int” assignments.

• Do not pass a pointer to a function that expects an int since this will result in lost information. For example, avoid assignments like the following:

```c
void f();
char *cp;
f(cp);
```

This nonportable function declaration will produce a compiler warning if you use ANSI C prototypes such as the following:

```c
void f(int);
char *cp;
f(cp);
```

Use the lint \(-Q\) command to find these “pointer-to-int” assignments.

• Use `void *type` if you must use a generic pointer type. This is preferable to converting a pointer to a type long.

• Beware of aliasing — multiple definitions of the same object. For example, the following two structures refer to the same object in different ways:

```c
struct node {
    int src_addr, dst_addr;
    char *name;
};

struct node {
    struct node *src, *dst;
    char *name;
}
```
Either replace this code with a union declaration, or change int to long. Be thorough when you port this type of code. The interdependencies and incompatibilities between two structures such as this can be difficult to find.

- Examine all assignments of a long to a double. Such assignments can result in a loss in accuracy.

On 32-bit platforms, a value of type double contains an exact representation of any value stored in a long (or a pointer) since a long is 32 bits and a double is 64 bits with 48 bits of mantissa. On a 64-bit platform, that is no longer a valid assumption.

9.5.3 The sizeof Operator

The result of the sizeof operator is of type size_t — an unsigned long on Alpha platforms.

9.5.4 Pointer Subtraction

The length of the integer required to hold the difference between two pointers to members of the same array (ptrdiff_t, defined in header file stddef.h) is a signed long on Alpha platforms.

9.6 Structures

The 64-bit data size of the long and pointer types affects the size, member alignment, alignment, and bit fields of structures.

9.6.1 Size

The size of structures and unions in Alpha applications can differ from those on 32-bit platforms. For example, the following structure, TextNode, doubles in size on a 64-bit platform because the pointer types are double in size (from 4 bytes to 8 bytes):

```c
struct TextNode{
    char *text;
    struct TextNode *left;
    struct TextNode *right;
};
```

If your program shares structures between 32- and 64-bit platforms, be careful about using variables of type long and pointer as members of those shared structures. These data types introduce sizes that are not available on 32-bit platforms.
If it is important that your program be completely portable, do the following:

- Use `typedef` types in structures and set up the types as appropriate for the platform. You can do this automatically by using information in the `limits.h` header file.
- Be careful when you build unions between variables of type `int` and `pointer` since they are not the same size.

### 9.6.2 Alignment

In the 64-bit environment, structures are aligned according to the most strictly aligned member. This aids in aligning other structure members on their required boundaries. Structures are also padded to ensure proper alignment. Padding can be added within the structure or at the end to terminate the structure on the same alignment boundary on which it started.

#### 9.6.2.1 Member Alignment

Members of structures and unions are said to be aligned on their natural boundaries. That is, `char` is aligned on a byte boundary, `short` on a word boundary, `int` on a longword boundary, and `long` and `pointer` on quadword boundaries.

This means that additional space can be used for padding member alignment in structures and unions. For example, on 32-bit platforms the size of the following structure is 16 bytes. On 64-bit Alpha platforms, the size of the structure is 32 bytes: 8 bytes for each pointer and 4 bytes of padding after the member, `size`, for the alignment of the pointer, `left`.

```c
struct TextCountNode {
    char *text;
    int size,
    struct TextCountNode *left;
    struct TextCountNode *right;
};
```

#### 9.6.2.2 Alignment Guidelines

Observe the following alignment guidelines when you work with structures in a 64-bit environment:

- Always use the `sizeof` operator to determine the size of a structure. Do not assume the size of a structure is the accumulated size of all objects defined in it. Additional space might be taken up for padding the member alignment.
- Minimize the amount of padding needed in a structure by reordering the members.
9.6.2.3 Alignment Examples

In the following example, the size of CountedString is 16 bytes (text = 8 bytes, count = 4 bytes, tail padding = 4 bytes). This structure is aligned on a quadword boundary because the pointer requires quadword alignment. No additional padding (beyond 4 bytes of tail padding) is necessary because CountedString will naturally align on a quadword boundary.

```c
struct {
    char    *text;
    int     count;
} CountedString;
```

In the following example, including CountedString in the definition forces alignment of the beginning of the structure to be on a quadword boundary. Additional padding might be introduced (depending on the value of MAX_LINE) to ensure proper quadword alignment for the structure member, string. Additional padding might also be introduced at the end of the structure to ensure proper structure alignment for arrays of these structures.

```c
CountedString CsArray[10]
struct {
    char    line[MAX_LINE];
    struct CountedString string;
} TextAndString;
```

In the following example, the structure has a size of 24 bytes.

```c
struct s {
    int     count;
    struct s    *next;
    int      total;
}
```

If this structure is reordered, the structure now has a size of 16 bytes.

```c
struct s {
    struct s    *next;
    int         count;
    int         total;
}
```

9.6.3 Bit Fields

Bit fields are allowed on any integral type on Alpha platforms. (ANSI C requires bit field support only for int, signed int, and unsigned int types.) In a C declaration, if one bit field immediately follows another in a structure declaration, the second bit field is packed into adjacent bits of the former unit. Since long is 64 bits in length on Alpha platforms, adjacent declarations of bit fields of type long may contain multiple bit field definitions in cases that...
previously did not on 32-bit systems. This change may cause different results in operations on these bit fields.

To ensure the same behavior in operations on bit fields, change bit field definitions of type `long` to `int`.

### 9.7 Pointers

The most common problems you will encounter when porting your programs are caused by assignment statements based on the assumption that pointers are the same length as `int` variables.

C programs commonly assign pointers to `int` variables. Such assignments are not recommended, but they do produce correct results on platforms in which pointers and `int` variables are the same size. On a 64-bit platform, however, this practice can produce incorrect results because the high-order 32 bits of a 64-bit address are lost when a 64-bit pointer is assigned to a 32-bit `int` variable.

The following code example illustrates this problem on a 64-bit platform:

```c
{   
    char *x; /* 64-bit pointer */
    int z;  /* 32-bit int variable */
    
    x = malloc(1024); /* get memory, store addr in 64 bits */
    z = x;       /* assign low-order 32 bits of 64-bit pointer to 32-bit int variable */
}
```

You must recode your applications to correct any “pointer-to-int” assignments. Ideally, you should declare all pointers in your application to be of data type `pointer`, but `long` will also work. You can use the `lint -Q` command to find “pointer-to-int” assignments in existing source code.

The truncated address support option (`-taso`), which can help in some situations, will not work with applications under VxWorks for Alpha since the most significant 32 bits of all pointers in VxWorks applications are set and cannot be truncated.
9.8 File Systems

On 64-bit Alpha platforms, the file offset used to navigate within a file or file system (off_t) is defined as a 64-bit long. This gives the file systems supported under VxWorks for Alpha the capability to build much larger files and file systems than is possible on a 32-bit platform. It is possible, therefore, to build files and file systems that cannot be fully accessed by 32-bit platforms.

Keep this in mind when you work in a distributed environment that contains file systems shared between 32- and 64-bit platforms.

9.9 Unavailable Instructions

As a true RISC architecture, the Alpha architecture is biased toward very fast cycle times, a simple instruction set, and a deep pipeline. As a result, certain instructions are not provided on Alpha platforms.

• There is no integer divide instruction.
• There is no integer modulus (remainder) instruction — the C language “%” operator.
• There are no multi-byte string instructions.

VxWorks for Alpha provides generalized software routines for both division and remainder operations, but both these subroutines exact a price in execution time. If possible, rework your existing program to take advantage of one of the following:

• Instead of dividing by a constant, multiply by the reciprocal of the constant.
• Instead of dividing by a power of two, use the arithmetic shift operations.
• To find the remainder from a division by a power of two, mask out the high-order bits using the .AND. operation — the C language “&” operator.

9.10 Using the lint Command

The \-Q option of the lint command provides support for porting your programs by identifying the following programming techniques that might cause problems in a 64-bit environment:

• Pointer alignment
• Pointer and integer data type combinations
• Assignments that cause a truncation of long values
• Assignments of long data to other types
• Structure and pointer combinations
• Type castings
• Format control strings in \texttt{scanf()} and \texttt{printf()} calls

Be aware that \texttt{lint} may give you quite a bit of information about your VxWorks for Alpha application, some of which may not be pertinent to the problems with porting your application.

For more information about using \texttt{lint}, see the Digital UNIX \texttt{lint} manpage.
Using the 68K VMEbus Library Interface

To facilitate porting existing VxWorks device drivers from Motorola 68K target platforms, VxWorks for Alpha includes a “68K VMEbus” interface in the AXPvme board support package (BSP). Using this interface you can port your existing 68K VxWorks device drivers to an AXPvme platform with a minimum of programming effort.

To use this interface, you must have INCLUDE_VME_68K_INTERFACE defined in the yourCloneTree/config/AXPvme/config.h configuration file.

10.1 sysLib Routines

As described in Chapter 5, I/O support under VxWorks for Alpha is based on a hierarchy of adapter data structures and two layers of support software:

- A high-level layer of adapter interface routines that operate independently from platform type. These routines reside in the adpLib system library and are called by user code to connect devices into the I/O system, manage register mapping, and handle interrupts.

- A low-level layer of hardware-dependent code that handles the device-specific functions. These routines reside in the sysAdpLib library of the system’s BSP and provide the board-level underpinnings to support the adpLib routines.

The 68K VMEbus interface is a “jacket” interface. That is, the interface is consistent with the sysLib routines of the Motorola 68K board support packages, but the routines have been rewritten using adpLib to provide VMEbus I/O support as required by the VxWorks for Alpha software architecture.
The following **sysLib** routines have been adapted for use in the 68K VMEbus interface:

- `sysBusToLocalAdrs()` — convert a VMEbus address to a local address (for outbound (PIO) mapping)
- `sysLocalToBusAdrs()` — convert a local address to a VMEbus address (for inbound (DMA) mapping)
- `sysIntEnable()` — enable a VMEbus interrupt level
- `sysIntDisable()` — disable a VMEbus interrupt level
- `sysBusIntGen()` — generate a VMEbus interrupt
- `sysBusTas()` — test and set a location across the VMEbus

Note that since AXPvme platforms do not provide a hardware test and set (TAS) instruction, routine `sysBusTas()` is implemented in the software. This imposes some restrictions on its use. See the description of routine `sysBusTas()` in Section 10.6 for details.

### 10.2 New I/O Routines

As mentioned in Chapter 3, the Alpha architecture has a minimum access granularity of 32 bits and does not support hardware access of byte or word data types. To accommodate 8- and 16-bit I/O, Alpha processors implement a mechanism called sparse-space addressing. In this addressing mode, the bus address contains an encoded data type that the CPU uses to extract the data from a 32-bit bus read or write. See Chapter 3 for a discussion of sparse-space addressing.

Because a sparse-space address is not a true bus address, conventional pointer I/O (as defined in Section 5.5.2) is not possible in sparse space. Therefore, the 68K VMEbus interface provides the following routines and macros for doing pointer I/O. Each routine has a corresponding macro, shown in square brackets:

**Write routines**

- `sysVmeWriteByte()` [VME_WRITE_BYTE()] — 8-bit write
- `sysVmeWriteWord()` [VME_WRITE_WORD()] — 16-bit write
- `sysVmeWriteInt()` [VME_WRITE_INT()] — 32-bit write
- `sysVmeWriteQuad()` [VME_WRITE_QUAD()] — 64-bit write

**Read routines**

- `sysVme.ReadByte()` [VME_READ_BYTE()] — 8-bit read
Using a macro is equivalent to calling the corresponding routine, except that the macro does not return status information. The macros generate inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototypes of these macros.

To port your 68K device driver, replace all pointer I/O operations to VMEbus sparse-space addresses with the appropriate routine listed above. These routines expect an address returned by routine sysBusToLocal() as input. They handle all the data and address manipulations required for sparse-space addressing. They also let you access contiguous locations by using the unary increment (++) and decrement (--) operators as shown in the following example:

```c
char *bytePtr;
int vmeAddress = 0x100;
sysBusToLocalAdrs(VME_AM_STD_SUP_PGM, vmeAddress, &bytePtr);
sysVmeWriteByte(bytePtr++, 1);
sysVmeWriteByte(bytePtr, 2);
```

Note that all addresses returned from sysBusToLocalAdrs() — even those mapped in sparse space — can be manipulated using the unary increment and decrement operators.

Bus addresses mapped in dense space can be accessed either by using normal pointer I/O or by supplying the bus address to sysVmexxxxInt() or sysVmexxxxQuad().

See Section 10.5 for more information about address mapping.

10.3 Probing the Bus

The VxWorks routine memProbe() is used to probe an address for a bus error. However, memProbe() has no way of knowing that the address presented to it is a VMEbus mapped address. Therefore, the 68K VMEbus interface includes a new routine, sysVmeMemProbe(), that handles only VMEbus addresses.

To port your driver, replace any memProbe() calls to VMEbus addresses with calls to sysVmeMemProbe().
10.4 Connecting to Bus Interrupts

When porting your 68K device driver, you can continue to use routine `intConnect()` to connect your driver's interrupt service routines to hardware interrupts, subject to the following restriction:

Unlike 68K platforms, the AXPvme has three separate sources of interrupts, rather than just one:

- I/O interrupts from the UARTs, watchdog timers, and the PCI A, B, C, D interrupts, through the Intel SIO bridge
- I/O interrupts from the VMEbus through the VMEbus interface connector (VIC)
- non-I/O machine checks

Connecting to an interrupt requires that the interrupt vector include its source, as well as its vector offset. Therefore, the 68K VMEbus interface redefines the `INUM_TO_IVEC` macro to encode the VMEbus adapter ID into the interrupt vector.

Note that this means this macro will no longer work for other, non-VMEbus, interrupts. If your driver connects to other interrupt sources, you must use routine `adpIntConnect()`, specifying the vector with a unique vector descriptor that identifies the adapter structure, as well as the vector offset. (See the VxWorks Reference Manual for a detailed description of the `adpIntConnect()` routine.)

The new definition of the `INUM_TO_IVEC` macro is included in the `yourCloneTree/config/src/AXPvme/AXPvme.h` header file.

10.5 Mapping the Bus

The AXPvme permits dynamically configured scatter/gather registers for VMEbus address mapping. On the other hand, 68K platforms use static mapping for both inbound and outbound I/O. To simulate this static mapping, the 68K VMEbus interface configures the inbound and outbound windows during system initialization.

10.5.1 Outbound Mapping

Since there are only a limited number of VMEbus outbound mapping registers available, the 68K VMEbus interface implements the following mapping scheme for outbound (PIO) transactions:

- A16 space is mapped entirely. To provide 8-, 16-, 32-, and 64-bit access granularity, it is mapped in sparse space.
68K VMEbus Interface Routines

- Except for the AXPvme's own A24 inbound window, all of A24 space is also mapped in sparse space.
- Approximately one-half the remaining mapping resources (about 200 Mbytes) are used to map A32 space. It is mapped to dense space, allowing only 32- and 64-bit access.

By limiting the amount of statically mapped A32 space, outbound mapping resources remain available for other drivers that depend on the dynamic mapping of the adpLib routines. Both the amount of space mapped and the A32 mapped base address are configurable, as follows:

- VME_68K_A32_OUTBND_WINDOW_SIZE specifies the total size of A32 space to be mapped for outbound transactions.
- VME_68K_A32_OUTBND_ADDRESS_BASE specifies the base A32 space address at which to begin mapping.

Note that any address space used for inbound mapping is not mapped for outbound use. Attempting to access inbound address space for outbound transfers will result in unpredictable results.

10.5.2 Inbound Mapping

To map inbound (DMA) transactions, the 68K VMEbus interface duplicates the 68K method of statically configuring address modifiers and swap modes during initialization. All available operating system memory is mapped to the VMEbus using A32 space.

With this mapping scheme, device drivers using adpLib routines on a system where the 68K VMEbus interface is enabled can find themselves starved of A32 inbound mapping resources. On systems with 128 Mbytes of memory, A32 inbound mapping registers are especially scarce. However, all A24 inbound mapping registers remain available for use with the adpLib interface.

Note also that the inbound base address for the AXPvme is set, by default, to the VMEbus address equal to two times SYSTEM_MEMORY_SIZE in A32 space. You must make sure that both the VME_A32_BASE and the VME_A32_SIZE console environment variables or the macros in config.h are set to appropriate values.

10.6 68K VMEbus Interface Reference

This section contains reference pages for the 68K VMEbus interface routines.
**68K VMEbus Interface Routines**

---

**sysBusIntGen()**

Generate a bus interrupt

**C Syntax**

```c
#include <sysLib.h>
STATUS sysBusIntGen ( int level,
                      int vector);
```

**Arguments**

- **level**
  Specifies the VMEbus interrupt level to generate. Valid values are 1 through 7.

- **vector**
  Specifies the interrupt vector to generate. Valid values are 0 through 255.

**Description**

This routine generates a VMEbus interrupt for a specified interrupt level.

**Return Value**

- OK if successful.
- ERROR if the value of level or vector is out of range.

---

**sysBusTas()**

Test and set a location across the bus

**C Syntax**

```c
#include <sysLib.h>
BOOL sysBusTas ( char *adrs);
```
68K VMEbus Interface Routines

Argument

$\textit{adrs}$

Specifies the address to be tested and set.

Description

Since the AXPvme has no hardware “test and set” (TAS) instruction, this routine performs a software “test and set” of the bus address.

______________________________ Warning ______________________________

This routine requires that the CPU have sole access to the VMEbus interface for outbound transfers. The user must guarantee that no PCI device is set up and enabled to read or write some device or memory on the VMEbus.

Also, because the PCI to VMEbus bridge does not support the PCI resource lock mechanism, the Digital AXPvme board cannot guarantee exclusive access to the VMEbus for multiple PCI masters.

Return Values

TRUE if the value was not previously set and now has been.
FALSE if the value was already set.

sysBusToLocalAdrs()

Convert a bus address to a local address

C Syntax

```c
#include <sysLib.h>
#include <vme.h>

STATUS sysBusToLocalAdrs(int adrsSpace, char *busAdrs, char **pLocalAdrs);
```
68K VMEbus Interface Routines

Arguments

*adrsSpace*
Specifies the bus address space in which `busAdrs` resides. (See `vme.h` for valid values.)

*busAdrs*
Specifies the bus address to convert.

*pLocalAdrs*
Receives the address of the local address.

Description

This routine gets the local address that accesses a specified VMEbus memory address.

If `adrsSpace` is either SHORT or STD, the address returned is an encoded sparse-space address. This address cannot be used for direct I/O. To access the bus location, the `pLocalAdrs` must be supplied as input to the `sysVmeReadxxxx()` or `sysVmeWritexxxx()` routines.

If `adrsSpace` is EXT, the address returned is a true dense-space address and can be used for either direct pointer I/O or as input to the `sysVmeReadxxxx()` or `sysVmeWritexxxx()` routines.

Return Values

OK if successful.

ERROR if the address space is unknown.

See Also

`sysLocalToBusAdrs()`
sysIntDisable()

Disable a bus interrupt level

C Syntax
#include <sysLib.h>
STATUS sysIntDisable (int intLevel);

Argument
intLevel
Specifies the interrupt level to disable. Valid values are 1 through 7.

Description
This routine disables a specified VMEbus interrupt level.

Return Values
OK if successful.
ERROR if the value of intLevel is not in the range 1 through 7.

See Also
sysIntEnable()

sysIntEnable()

Enable a bus interrupt level

C Syntax
#include <sysLib.h>
STATUS sysIntEnable (int intLevel);
68K VMEbus Interface Routines

Argument

\textit{intLevel}

Specifies the interrupt level to enable. Valid values are 1 through 7.

Description

This routine enables a specified VMEbus interrupt level.

Return Values

OK if successful.

ERROR if the value of \textit{intLevel} is not in the range 1 through 7.

See Also

\textit{sysIntDisable()}

\underline{sysLocalToBusAdrs()}

Convert a local address to a bus address

C Syntax

\begin{verbatim}
#include <sysLib.h>
#include <vme.h>

STATUS sysLocalToBusAdrs (int \textit{adrSpace},
                         char *\textit{localAdrs},
                         char **\textit{pBusAdrs});
\end{verbatim}

Arguments

\textit{adrSpace}

Specifies the bus address space in which \textit{pBusAdrs} resides. (See \textit{vme.h} for valid values.)

\textit{localAdrs}

Specifies the local address to convert.

\textit{pBusAdrs}

Receives the address of the bus address.
68K VMEbus Interface Routines

Description
This routine gets the VMEbus address that accesses a specified local memory address.

Return Values
OK if successful.
ERROR if the local address cannot be accessed from the bus.

See Also
sysBusToLocalAdrs()

sysVmeMemProbe()
Probes a VMEbus address for a bus error

C Syntax
#include <sysLib.h>

STATUS sysVmeMemProbe (char *adrs,
                        int mode,
                        int length,
                        char *pVal);

Arguments
adrs
Specifies the bus address to be probed. This is an address derived from the address previously returned from sysBusToLocalAdrs().

mode
Specifies whether to probe if the address is readable (VX_READ) or writable (VX_WRITE).

length
Specifies the length to be read or written: 1, 2, 4, or 8 bytes. Any other value will produce unpredictable results.
**68K VMEbus Interface Routines**

*pVal*

Specifies where to return the value for a READ probe, or a pointer to the value to be written for a WRITE probe.

**Description**

This routine probes a specified address to see if it is readable or writable. If the probe is a READ, the value read will be copied to the location pointed to by *pVal*. If the probe is a WRITE, the value written will be taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of 1, 2, 4, or 8 bytes, as specified by length.

Note that only bus errors are trapped during the probe, and that the access must otherwise be valid (that is, not generate an address error.)

**Return Values**

OK if the probe is successful.

ERROR if the probe caused a bus error or an address misalignment.

---

**sysVmeReadByte()**

Read 8 bits of data (one byte) from a bus address

**C Syntax**

```c
#include <sysLib.h>

STATUS sysVmeReadByte (void *adrs,
                         u_char *data);
```

**Arguments**

*adrs*

Specifies the bus address from which to read. This is an address derived from the address previously returned from `sysBusToLocalAdrs()`.

*data*

Specifies the address to receive the data.
Description

This routine reads one byte of data from the specified bus address and stores the value in a user-specified location. The bus address must be a sparse-space address.

Using the VME_READ_BYTE macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file `yourCloneTree/config/AXPvme/AXPvme.h` for the prototype of this macro.

Return Values

OK if successful.
ERROR if `adrs` is invalid.

```
sysVmeReadInt()
```

Read 32 bits of data (one integer) from a bus address

C Syntax

```
#include <sysLib.h>

STATUS sysVmeReadInt (void *adrs,
                      u_int *data);
```

Arguments

- `adrs`
  Specifies the bus address from which to read. This is an address derived from the address previously returned from `sysBusToLocalAdrs()`.

- `data`
  Specifies the address to receive the data.
68K VMEbus Interface Routines

Description

This routine reads one integer (32 bits of data) from the specified bus address and stores the value in a user-specified location. The bus address can be either a sparse- or a dense-space address.

Using the VME_READ_INT macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype of this macro.

Return Values

OK if successful.
ERROR if adrs is invalid.

sysVmeReadQuad()

Read 64 bits of data (one quadword) from a bus address

C Syntax

#include <sysLib.h>

STATUS sysVmeReadQuad (void *adrs,
                       u_quad *data);

Arguments

adrs
Specifies the bus address from which to read. This is an address derived from the address previously returned from sysBusToLocalAdrs().

data
Specifies the address to receive the data.
68K VMEbus Interface Routines

Description

This routine reads one quadword (64 bits of data) from the specified bus address and stores the value in a user-specified location. The bus address can be either a sparse- or dense-space address.

Using the VME_READ_QUAD macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype of this macro.

Return Values

OK if successful.
ERROR if adrs is invalid.

sysVmeReadWord()

Read 16 bits of data (one word) from a bus address

C Syntax

#include <sysLib.h>

STATUS sysVmeReadWord (void *adrs, u_word *data);

Arguments

\textit{adrs}  
Specifies the bus address from which to read. This is an address derived from the address previously returned from \texttt{sysBusToLocalAdrs()}.  

\textit{data}  
Specifies the address to receive the data.
68K VMEbus Interface Routines

Description
This routine reads one word (16 bits of data) from the specified bus address and stores the value in a user-specified location. The bus address must be a sparse-space address.

Using the VME_READ_WORD macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype of this macro.

Return Values
  OK if successful.
  ERROR if adrs is invalid.

sysVmeWriteByte()
Write 8 bits of data (one byte) to a bus address

C Syntax
#include <sysLib.h>
STATUS sysVmeWriteByte (void *adrs,
    u_char data);

Arguments
  adrs
  Specifies the bus address to which to write. This is an address derived from the address previously returned from sysBusToLocalAdrs().

  data
  Specifies the data to be written.
68K VMEbus Interface Routines

Description
This routine writes one byte of user-supplied data to the specified bus address. The bus address must be a sparse-space address.

Using the VME_WRITE_BYTE macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype of this macro.

Return Values
OK if successful.
ERROR if adrs is invalid.

sysVmeWriteInt()

Write 32 bits of data (one integer) to a bus address

C Syntax
#include <sysLib.h>
STATUS sysVmeWriteInt (void *adrs,
                       u_int data);

Arguments
adrs
Specifies the bus address to which to write. This is an address derived from the address previously returned from sysBusToLocalAdrs().

data
Specifies the data to be written.

Description
This routine writes one user-supplied integer (32 bits of data) to the specified bus address. The bus address can be either a sparse- or a dense-space address.

Using the VME_WRITE_INT macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype of this macro.
68K VMEbus Interface Routines

Return Values

OK if successful.
ERROR if adrs is invalid.

sysVmeWriteQuad()

Write 64 bits of data (one quadword) to a bus address

C Syntax

#include <sysLib.h>

STATUS sysVmeWriteQuad ( void *adrs,
                         u_quad data );

Arguments

adrs
Specifies the bus address to which to write. This is an address derived from
the address previously returned from sysBusToLocalAdrs().

data
Specifies the data to be written.

Description

This routine writes one user-supplied quadword (64 bits of data) to the
specified bus address. The bus address can be either a sparse- or dense-space
address.

Using the VME_WRITE_QUAD macro is equivalent to calling this routine,
except that the macro does not return status information. The macro generates
inline code. See file yourCloneTree/config/AXPvme/AXPvme.h for the prototype
of this macro.

Return Values

OK if successful.
ERROR if adrs is invalid.
sysVmeWriteWord()

Write 16 bits of data (one word) to a bus address

C Syntax

```c
#include <sysLib.h>
STATUS sysVmeWriteWord (void *adrs, 
                      u_word data);
```

Arguments

- **adrs**
  Specifies the bus address to which to write. This is an address derived from the address previously returned from `sysBusToLocalAdrs()`.

- **data**
  Specifies the data to be written.

Description

This routine writes one user-supplied word (16 bits of data) to the specified bus address. The bus address must be a sparse-space address.

Using the VME_WRITE_WORD macro is equivalent to calling this routine, except that the macro does not return status information. The macro generates inline code. See file `yourCloneTree/config/AXPvme/AXPvme.h` for the prototype of this macro.

Return Values

- **OK** if successful.
- **ERROR** if `adrs` is invalid.
This chapter contains two examples of how to port an existing VxWorks driver from a 68K target to an Alpha platform.

- Example 11–1 in Section 11.1 shows selected code taken from the CMC ENP–10/L Ethernet network interface driver. This driver was ported to the Digital AXPvme board using the adpLib library routines where appropriate.
- Example 11–2 in Section 11.2 shows the same selection of code ported using the 68K VMEbus interface, as described in Chapter 10.

The CMC ENP–10/L Ethernet driver provides one user-callable routine, `enpattach()`, shown in the two examples. This routine publishes the ENP interface by creating a network interface record and adding this record to the system list. The routine also initializes the driver and the device.

For the complete source listing of this driver, see file `if_enp.c` in the `yourCloneTree/src/demo/AXPenp` directory.

The callouts in the examples mark sections of the code where changes were made in order to port the driver. The callouts correspond to items in the explanation that follows each example.
11.1 Porting Example Using the adpLib Library

Example 11–1 Porting with the adpLib Library

```c
/* include files */
#if (CPU_FAMILY == ALPHA)
#include "adpLib.h"
#include "config/AXPvme/AXPvme.h"
#endif

/* The definition of the driver control structure */
typedef struct drv_ctrl
{
    IDR idr; /* interface data record */
    BOOL attached; /* TRUE if attach() has succeeded */
    ENPDEVICE *enpAddr; /* VMEbus address of board */
    ENPSTAT enpStat; /* duplicate copy of ENP statistics */
#if (CPU_FAMILY == ALPHA)
    UINT lostCarrier; /* statistic counter */
#else
    ULONG lostCarrier; /* statistic counter */
#endif
    int enpIntLevel; /* VMEbus interrupt level */
    int enpIntVec; /* VMEbus interrupt vector */
    BOOL taskLevelActive; /* netTask processing packets */
    SEM_ID TxSem; /* transmitter semaphore */
} DRV_CTRL;

/* VME mapping structure */
#if (CPU_FAMILY == ALPHA)
typedef struct {
    unsigned long map_base;
    unsigned long vme_granularity;
    unsigned long vme_map_size;
    struct adpMapObj *vme_map_handle;
    struct adp *srcAdapter;
    struct adp *dstAdapter;
} VME_MAP_INFO;
LOCAL VME_MAP_INFO enp_map;
#endif

(continued on next page)
Example 11–1 (Cont.) Porting with the adpLib Library

/*
 * enpattach - publish the interface, and initialize the driver and
 * device
 */

STATUS enpattach
{
    int unit, /* unit number */
    char *addr, /* address of ENP’s shared memory */
    int ivec, /* interrupt vector to connect to */
    int ilevel, /* interrupt level */
    int enpAddrAm /* address VMEbus address modifier*/
}

    int delay;
    short status;
    int temp;
    ENPDEVICE *pEnpDev;
    long enpBase;
    DRV_CTRL *pDrvCtrl;
    unsigned long offset;

    /* Publish the interface data record */
    ether_attach(
        & pDrvCtrl->idr.ac_if, unit,
        "enp", /* interface name */
        (FUNCPTR) NULL, /* no init() routine */
        (FUNCPTR) enpIoctl,
        (FUNCPTR) enpOutput,
        (FUNCPTR) enpReset
    );

    /* Create the transmit semaphore. */
    pDrvCtrl->TxSem = semMCreate (
        SEM_Q_PRIORITY |
        SEM_DELETE_SAFE |
        SEM_INVERSION_SAFE
    );

    if (pDrvCtrl->TxSem == NULL)
    {
        printf ("enp: error creating transmitter semaphore\n");
        return (ERROR);
    }

    /* Determine local address and AM codes to access controller
    On Alpha, create a VMEbus to Local map */
    #if (CPU_FAMILY == ALPHA)
        (continued on next page)
Example 11–1 (Cont.) Porting with the adpLib Library

```c
enpBase = (long)addr;
enp_map.srcAdapter = NULL;
enp_map.dstAdapter = adpLookup(VME_VECTOR(0,0));
enp_map.map_base = -1;
if (adpCreateMapPath(enp_map.srcAdapter, /* local */
enp_map.dstAdapter, /* VMEbus */
(unsigned long)addr, /* ENP Base adrs */
0x20000, /* sz of ENP adrs spc */
0,
amToAdp(enpAddrAm) | /* adp eqv of adrs mod */
ADDRESS_SPACE_SPARSE, /* Sparse space */
(long)ACCESS_TYPE_NO_SWAP,/* Byte swapping is not used. We use network routines. */
&enp_map.map_base, /* returned base adrs of ENP in process space */
&enp_map.vme_map_handle, /* returned io handle */
&enp_map.vme_map_size, /* size of map */
&enp_map.vme_granularity /* for offset calc */
) == ERROR)
#else
if (sysBusToLocalAdrs (enpAddrAm, addr, &enpBase) == ERROR)
#endif
{
errnoSet (EFAULT);
return (ERROR);
}
#endif
pEnpDev = (ENPDEVICE *) enpBase;
enpBase += 0x1000; /* memory is offset from board base address */
5 /* Connect the interrupt handler */
#if (CPU_FAMILY == ALPHA)
adpIntConnect((VOIDFUNCPTR)VME_VECTOR(0,ivec), (long)unit,
#else
(void) intConnect ((VOIDFUNCPTR *)INUM_TO_IVEC (ivec),
 (VOIDFUNCPTR)enpIntr, unit);
#endif
/* wait for kernel reset to finish */
delay = 15 * sysClkRateGet ();
```

(continued on next page)
Example 11–1 (Cont.) Porting with the adpLib Library

```c
while (1)
{
  /* Test for response on the bus */
  #if (CPU_FAMILY == ALPHA)
    offset =
      ((unsigned long) &pEnpDev->enp_status % enp_map.vme_granularity);
    if ( adpProbeBus(enp_map.dstAdapter,
       enp_map.map_base,
       offset,
       VX_READ,
       2,
       &status) != ERROR)
  #else
    if ( vxMemProbe ((char *) & pEnpDev->enp_status,
                     O_RDONLY,
                     sizeof (short),
                     (char *) & status
       ) == OK)
  #else
    }
  /* Check status word */
  if ( (ntohs (status) & STS_READY) != 0 )
    break;
  }
  /* Check for timeout */
  if (--delay <= 0)
  {
    (void) errnoSet (S_iiosLib_CONTROLLER_NOT_PRESENT);
    return (ERROR);
  }
  taskDelay (sysClkRateGet() / 30); /* 1/30th of a second */
}
```

(continued on next page)
Example 11–1 (Cont.) Porting with the adpLib Library

/* start link level firmware */
#if (CPU_FAMILY == ALPHA)
    axpVmeWrite16(&enp_map, (char *) &pEnpDev->enp_intvector, htons(ivec));
#else
    pEnpDev->enp_intvector = htons (ivec);
#endif
#if (CPU_FAMILY == ALPHA)
    temp = htonl ((unsigned int) enpBase);
    WRITELONG ( (char *) &pEnpDev->enp_base, temp);
#else
    temp = htonl ((unsigned int) enpBase);
    WRITELONG ( (u_long) &pEnpDev->enp_base, temp);
#endif
#if (CPU_FAMILY == ALPHA)
    axpVmeWrite16(&enp_map, (char *) &pEnpDev->enp_go, htons(GOVAL));
#else
    pEnpDev->enp_go = htons (GOVAL);
#endif
CACHE_PIPE_FLUSH ();
/* wait for the link-level firmware to finish initialization */
delay = 15 * sysClkRateGet () ;
#if (CPU_FAMILY == ALPHA)
    while ( (ntohs (axpVmeRead16(&enp_map, (char *) &pEnpDev->enp_state)))
        & S_ENPRUN) == 0)
#else
    while ( (ntohs (pEnpDev->enp_state) & S_ENPRUN) == 0)
#endif
{  
    if ( --delay <= 0)  
    {  
        (void) errnoSet( S_ioLib_DEVICE_ERROR);
        return( ERROR);
    }
    taskDelay (sysClkRateGet() / 30); /* 1/30th of a second */
}
pDrvCtrl->enpAddr = pEnpDev;
pDrvCtrl->enpIntLevel = ilevel;
pDrvCtrl->enpIntVec = ivec;
enpgetaddr (unit);

(continued on next page)
Example 11–1 (Cont.) Porting with the adpLib Library

/* enable the interrupt */
#if (CPU_FAMILY == ALPHA)
   adpSetBusIntEna(adpLookup(VME_VECTOR(0,0)),(long)ivec,(long)ilevel);
#else
   (void) sysIntEnable (ilevel);
#endif

/* Raise the interface flags */
pDrvCtrl->idr.ac_if.if_flags |= IFF_UP | IFF_RUNNING | IFF_NOTRAILERS;

/* Set our success indicator */
pDrvCtrl->attached = TRUE;
return (OK);
}

Callouts in the following list correspond to the callouts in Example 11–1.

1 To use the adapter interface library, adpLib, you must include the following header files:
   • adpLib.h
   • config/AXPvme/AXPvme.h

2 On Alpha systems, long data types are 64 bits, so you should change long declarations to 32-bit int data types.

3 The VME_MAP_INFO structure is used in this driver to hold the VMEbus-to-process-space mapping information returned by routine adpMapCreatePath(). It contains the information needed by other utilities within the driver for device access and calculating address offsets.

4 The adpLib routine adpCreateMapPath() maps a range of VMEbus addresses into the program space. The parameters to adpCreateMapPath() and the returned mapping parameters are stored in the VME_MAP_INFO structure, enp_map.
   The routine amToAdp() converts a VMEbus address modifier to the proper adpLib constant.

5 Routine adpIntConnect() is used to connect an interrupt handler to its interrupt vector instead of routine intConnect().

6 Routine adpProbeBus() is used to probe the bus instead of routine vxMemProbe().
VMEbus accesses are done with adpLib routines adpRead() and adpWrite() rather than by pointers to VMEbus address space. Driver utility routines axpVmeWritexx() and axpVmeReadxx() are used here to simplify this interface. Note that these utilities use the enp_map structure which contains the VMEbus-to-process-space mapping parameters for the ENP 10 board.

For the AXPvme, the WRITELONG and READLONG macros are replaced by routines that perform the two 16-bit access. Note that since AXPvme long values are 64 bits while “network” long values are 32 bits, typecasts are required for routines expecting 32-bit values.

For drivers written for big endian devices, adpCreateMapPath() can be set to perform byte or word swapping. Alternately, this can be done in the software using htonl(), htons(), ntohl() and ntohs() for greater portability at the expense of efficiency.

Routine adpSetBusIntEna() is used to enable interrupts instead of routine sysIntEnable.
11.2 Porting Example Using the 68K VMEbus Interface

Example 11–2 Porting with the 68K VMEbus Interface

1 /* include files */
 ifndef (CPU_FAMILY == ALPHA)
 include "./if_enp.h"
 include "AXPvme.h"
 else
 include "drv/netif/if_enp.h"
 endif

 /* The definition of the driver control structure */
type define struct drv_ctrl
 {
  IDR idr; /* interface data record */
  BOOL attached; /* TRUE if attach() has succeeded */
  ENPDEVICE *enpAddr; /* VME address of board */
  ENPSTAT enpStat; /* duplicate copy of ENP statistics */
 ifndef (CPU_FAMILY == ALPHA)
   UINT lostCarrier; /* statistic counter */
 else
   ULONG lostCarrier; /* statistic counter */
 endif
 int enpIntLevel; /* VME interrupt level */
 int enpIntVec; /* VME interrupt vector */
 BOOL taskLevelActive; /* netTask processing packets */
 SEM_ID TxSem; /* transmitter semaphore */
 } DRV_CTRL;

 /* Local definitions */
 ifndef (CPU_FAMILY == ALPHA)
 LOCAL u_long savedEnpAddr;
 define FIX_ADDRESS(a) {u_long}(savedEnpAddr + {u_long}a & 0xffffffff)
 endif

 */

 * enpattach - publish the interface, and initialize the driver
 * and device
 */

(continued on next page)
Example 11–2 (Cont.) Porting with the 68K VMEbus Interface

```c
STATUS enpattach
{
  int unit,     /* unit number */
  char *addr,   /* address of enp's shared memory */
  int ivec,     /* interrupt vector to connect to */
  int ilevel,   /* interrupt level */
  int enpAddrAm /* address VMEbus address modifier*/
}
{
  int delay;
  short status;
  int temp;
  ENPDEVICE *pEnpDev;
  long enpBase;
  DRV_CTRL *pDrvCtrl;
  unsigned long offset;
  /* Safety check unit number */
  if (unit < 0 || unit >= MAX_UNITS)
  {
    errnoSet (S_iosLib_CONTROLLER_NOT_PRESENT);
    return (ERROR);
  }
  /* Get pointer to our driver control structure */
  pDrvCtrl = &drvCtrl[unit];
  /* Check for multiple invocation per unit */
  if (pDrvCtrl->attached)
    return (OK);
  /* Publish the interface data record */
  ether_attach {
    & pDrvCtrl->idr.ac_if, unit,
    "enp", /* interface name */
    (FUNCPTR) NULL,    /* no init() routine */
    (FUNCPTR) enpioctl,
    (FUNCPTR) enpOutput,
    (FUNCPTR) enpReset
  };
  /* Create the transmit semaphore. */
  pDrvCtrl->TxSem = semCreate
  {
    SEM_Q_PRIORITY |
    SEM_DELETE_SAFE |  
    SEM_INVERSION_SAFE
  };
}
```

(continued on next page)
Example 11–2 (Cont.) Porting with the 68K VMEbus Interface

```c
if (pDrvCtrl->TxSem == NULL) {
    printf("enp: error creating transmitter semaphore\n");
    return (ERROR);
}

/* Determine local adrs and AM codes to access controller */
if (sysBusToLocalAdrs (enpAddrAm, addr, &enpBase) == ERROR) {
    errnoSet (EFAULT);
    return (ERROR);
}

pEnpDev = (ENPDEVICE *) enpBase;
#if (CPU_FAMILY == ALPHA)
savedEnpAddr = enpBase;
#endif
enpBase += 0x1000; /* memory is offset from board base address */

/* Connect the interrupt handler */
(void) intConnect ((VOIDFUNCPTR *)INUM_TO_IVEC (ivec),
    (VOIDFUNCPTR)enpIntr, unit);

/* wait for kernel reset to finish */
delay = 15 * sysClkRateGet ();
while (1) {

/* Test for response on the bus */
#if (CPU_FAMILY == ALPHA)
    if (sysVmeMemProbe((char *) & pEnpDev->enp_status, 
        O_RDONLY, 
        sizeof (short), 
        (char *) & status) == OK)
#else
    if (vxMemProbe ((char *) & pEnpDev->enp_status, 
        O_RDONLY, 
        sizeof (short), 
        (char *) & status) == OK)
#endif
    /* Check status word */
    if ( (ntohs (status) & STS_READY) != 0 )
        break;
}
```

(continued on next page)
Example 11–2 (Cont.) Porting with the 68K VMEbus Interface

```c
/* Check for timeout */
if (--delay <= 0)
{
    (void) errnoSet (S_iosLib_CONTROLLER_NOT_PRESENT);
    return (ERROR);
}
taskDelay (sysClkRateGet() / 30); /* 1/30th of a second */

/* start link level firmware */
#if (CPU_FAMILY == ALPHA)
    sysVmeWriteWord(FIX_ADDRESS((char *)&(pEnpDev->enp_intvector)),
                    htons(ivec));
#else
    pEnpDev->enp_intvector = htons (ivec);
#endif
#if (CPU_FAMILY == ALPHA)
    temp = htonl ((unsigned int) enpBase);
    WRITELONG ((char *)&pEnpDev->enp_base, temp);
#else
    temp = htonl ((u_long) enpBase);
    WRITELONG (&pEnpDev->enp_base, temp);
#endif
#if (CPU_FAMILY == ALPHA)
    sysVmeWriteWord(FIX_ADDRESS((char *)&pEnpDev->enp_go), htons(GOVAL));
#else
    pEnpDev->enp_go = htons (GOVAL);
#endif
CACHE_PIPE_FLUSH ();
/* wait for the link level firmware to finish initialization */
delay = 15 * sysClkRateGet () ;
#if (CPU_FAMILY == ALPHA)
    while ((ntohs (axpVmeRead16((char *) &(pEnpDev->enp_state))) & S_ENPRUN) == 0)
#else
    while ((ntohs (pEnpDev->enp_state) & S_ENPRUN) == 0)
#endif
{
    if (--delay <= 0)
    {
        (void) errnoSet (S_ioLib_DEVICE_ERROR);
        return (ERROR);
    }
}
```

(continued on next page)
Example 11–2 (Cont.) Porting with the 68K VMEbus Interface

```c
    taskDelay (sysClkRateGet() / 30); /* 1/30th of a second */
```

```c
    pDrvCtrl->enpAddr = pEnpDev;
    pDrvCtrl->enpIntLevel = ilevel;
    pDrvCtrl->enpIntVec = ivec;
    enpgetaddr (unit);
    (void) sysIntEnable (ilevel);
    /* Raise the interface flags */
    pDrvCtrl->idr.ac_if.if_flags |= IFF_UP | IFF_RUNNING | IFF_NOTRAILERS;
    /* Set our success indicator */
    pDrvCtrl->attached = TRUE;
    return (OK);
```

**Callouts in the following list correspond to the callouts in Example 11–2.**

1. To use the 68K VME bus interface, you must include the AXPvme.h header file.
2. On Alpha systems, long data types are 64 bits, so you should change long declarations to 32-bit int data types.
3. savedEnpAddr is used to store the original mapped address returned from routine sysBusToLocalAdrs(). Macro FIX_ADDRESS uses savedEnpAddr to ensure that addresses passed to the sysVmeWritexx and sysVmeReadxx routines have their 32 high bits intact.
4. Routine sysBusToLocalAdrs() can be used without modification. Within the 68K VMEbus interface, it has been adapted to handle VMEbus addressing. (See Section 10.1.)
5. The mapped address returned from routine sysBusToLocalAdrs() is saved in savedEnpAddr for later use by the FIX_ADDRESS macro. (See the third callout, above.)
6. Routine intConnect() can be used because we are connecting to a VMEbus interrupt. (See Section 10.4.)
7. Routine sysVmeMemProbe() is a new routine within the 68K VMEbus interface designed to handle probes to VMEbus addresses. (See Section 10.3.)
Under the 68K VMEbus interface, bus accesses are done with a set of new I/O routines — `sysVmeReadxx()` and `sysVmeWritexx()` — rather than by pointers to the VMEbus address space. These routines are written to accept addresses returned from routine `sysBusToLocalAdrs()` and handle all data and address manipulations required for sparse- and dense-space addressing. (See Section 10.2.)

For the AXPvme, the `WRITELONG` and `READLONG` macros are replaced by routines that perform the two 16-bit access. Note that since AXPvme long values are 64 bits while "network" long values are 32 bits, typecasts are required for routines expecting 32-bit values.

Routine `sysIntEnable()` can be used without modification. Within the 68K VMEbus interface, it has been adapted to handle VMEbus addressing. (See Section 10.1.)
Index

A

Adapter
  hardware description, 1-3
Adapter data structure
  See Data structures, adapter data structure
adpCheckInterrupt() routine, 5-40
adpClearInterrupt() routine, 5-40
adpCreateMapPath() routine, 3-4, 5-24, 5-27
adpFreeMapPath() routine, 3-4, 5-24
adpIntConnect() routine, 5-26, 10-4
adpLib library, 3-1, 4-4, 5-2, 5-24, 5-40, 8-34, 10-1, 11-1
adpPostInterrupt() routine, 5-40
adpRead() routine, 3-3
adpReset() routine, 5-40
adpRMW() routine, 5-40
adpWrite() routine, 3-3, 3-4, 3-5
Alpha architecture
  bus hierarchy, 3-1
  bus mapping
    See I/O addressing
data granularity, 1-7, 3-2, 3-3, 9-2, 9-3
  order of read and write operations, 1-7, 3-2, 3-4, 9-3
write buffer, 1-7, 3-2, 3-4
AXPpci target platform, 2-2
AXPvme target platform, 2-2, 5-23

B

bcopy_from_mbufs() macro, 7-21
bcopy_to_mbufs() routine, 7-22
Block devices
  See Device, block
Board support package (BSP), 3-1, 5-42
build_cluster() routine, 7-23
Bus, 1-2
  adapter hardware, 1-3
  resetting, 5-40
  address space, 1-3
  communications, 1-2
  I/O, 1-2
  Industry Standard Architecture (ISA), 2-3
  internal, 1-2
  interrupts
    posting, checking and clearing, 5-40
  local, 1-2
  mapping
  See I/O addressing
  Peripheral Component Interconnect (PCI), 2-3
  primary, 1-2
  secondary, 1-2
  VMEbus, 2-4
Bus interrupts
  posting, checking and clearing, 5-40
Bus-specific features, 5-40
Character devices

See Device, nonblock

check_trailer() routine, 7–25
CMC ENP-10/L Ethernet driver, 11–1
Communication, 5–27
configAll.h configuration file, 5–44
Configuring a driver, 5–43
  block devices, 5–43
  nonblock devices, 5–43
Control/status registers (CSRs), 1–3
copy_from_mbufs() macro, 7–21
copy_to_mbufs() macro, 7–22

Data structures, 5–3
  adapter data structure, 3–1, 5–2
  block device header (BLK_DEV), 5–3
  device descriptor, 5–3
  nonblock device header (DEV_HDR), 5–3
  SEL_WAKEUP_LIST, 5–21
  SEL_WAKEUP_NODE, 5–21
DECchip 21040 Ethernet driver, 7–36
Dense-space addressing
  See I/O addressing, dense space
Device
  block, 1–3, 4–2
  controller, 1–3
  control/status register (CSR), 1–3
  differences between block and nonblock, 4–2, 5–3
initialization
  See Initialization, devices
interrupt service routines (ISRs)
  See Interrupt Service Routines (ISRs)
multiple accesses
  See select() function
nonblock, 1–3, 4–2
nonstandard, 4–4
routines

Device routines (cont’d)
  See Driver routines
timeouts
  See select() function
Device controller, 1–3
Device descriptor
  See Data structures, device descriptor
Direct memory access (DMA), 1–4, 4–3, 5–27, 10–5
dosFS file system
  See File systems
do_protocol() routine, 7–26
do_protocol_with_type() routine, 7–26
Driver examples
  CMC ENP-10/L Ethernet driver, 11–1
  DECchip 21040 Ethernet driver, 7–36
  NCR 53C810 PSIOP SCSI driver, 8–34, 8–62
  NCR 53C90 ASC SCSI driver, 8–34
  NSC PC87311/PC87312 floppy disk driver, 5–2
  XYCOM 203 realtime clock driver, 4–5
  Z8C530 serial controller driver, 6–13
Driver routines
  block devices, 1–7, 5–10
    xxBlkCreate(), 5–38, 5–44
    xxBlkRd(), 5–11
    xxBlkWrt(), 5–13
    xxInit(), 5–34, 5–44
    xxIoctl(), 5–15, 5–21
    xxReset(), 5–18
    xxStatusChk(), 5–19
  network devices
    xxattach(), 7–12
    xxInit(), 7–14
    xxInt(), 7–15
    xxIoctl(), 7–16
    xxOutput(), 7–17
    xxReset(), 7–18
    xxWatchdog(), 7–18
  nonblock devices, 1–6, 5–4
    xxClose(), 5–8
    xxCreat(), 5–6
    xxDevCreate(), 5–32, 5–43
Driver routines
nonblock devices (cont’d)
  xxDrv(), 5–31, 5–43
  xxIoctl(), 5–10, 5–21
  xxOpen(), 5–7
  xxRead(), 5–9
  xxRemove(), 5–7
  xxWrite(), 5–9

SCSI devices
  xxxBusPhaseGet(), 8–17
  xxxBytesIn(), 8–18
  xxxBytesOut(), 8–19
  xxxCtrlCreate(), 8–20
  xxxCtrlInit(), 8–22
  xxxDevSelect(), 8–23
  xxxDmaBytesIn(), 8–24
  xxxDmaBytesOut(), 8–25
  xxxIntr(), 8–26
  xxxMsgInAck(), 8–27
  xxxScsiBusReset(), 8–28
  xxxScsiTransact(), 8–28
  xxxSelTimeOutCvt(), 8–30
  xxxShow(), 8–31

serial devices
  ttyCoDevCreate(), 6–7
  ttyCoDrv(), 6–9
  ttyCoInt(), 6–9
  ttyCoIoctl(), 6–10
  ttyCoOpen(), 6–11
  ttyCoStartup(), 6–12

E
EB64+ target platform, 2–3
EB64 target platform, 2–2, 5–23
EB66 target platform, 2–3
etherInputHookRtn() routine, 7–6
etherLib library, 7–6
etherOutputHookRtn() routine, 7–7
etherAttach() routine, 7–28
ether_output() routine, 7–29

F
Files
  for accessing devices, 1–4
  File systems, 1–5, 5–38, 5–44
  Floating-point instructions
    use in ISRs, 5–26
fppALib library, 5–26
free() routine, 5–26

I
I/O addressing, 1–7, 3–2, 3–3, 5–23
  by dereferencing pointers, 5–24
  dense space, 3–3, 5–24, 10–3
  direct memory access
    See direct memory access (DMA)
  map base and offset, 5–24
  map granularity, 5–24
  map registers, 1–4
  minimum VMEbus map granularity, 5–24
  scatter/gather mapping, 1–4, 3–3, 4–3, 5–23
  sparse space, 3–3, 5–24, 10–2
I/O address space, 1–3
I/O requests
  See I/O system, requests to
I/O system, 1–4
  files, 1–4
  file systems
    See File systems
  requests to, 1–5, 1–6
I/O transfers
  direct memory access
    See Direct memory access (DMA)
  programmed I/O (PIO)
    See Programmed I/O (PIO)
IF_DEQUEUE() macro, 7–30
IF_ENQUEUE() macro, 7–31
Inbound I/O
  see Direct memory access (DMA)
Industry Standard Architecture (ISA) bus
   See ISA bus
Initialization, 5–31
devices, 1–6
   block, 5–38
   nonblock, 5–32
drivers, 1–6
   block devices, 5–34
   nonblock devices, 5–31
file systems, 5–44
intConnect() routine, 10–4
Interrupt service routines (ISRs), 1–6, 5–25
   connecting to, 5–26
   exception conditions, 5–26
   restrictions, 5–25
   use of floating-point instructions, 5–26
   use of semaphores, 5–25
INUM_TO_IVEC macro, 10–4
iosDevAdd() routine, 5–3, 5–32
iosDrvInstall() routine, 5–32
iosInit() routine, 5–43
ISA bus, 2–3

68K VMEbus interface, 10–1
   connecting to interrupts
      intConnect(), 10–4
   I/O addressing, 10–4
      inbound mapping, 10–5
      outbound mapping, 10–4
modified sysLib routines
   sysBusIntGen(), 10–2, 10–6
   sysBusTas(), 10–2, 10–6
   sysBusToLocalAdrs(), 10–2, 10–7
   sysIntDisable(), 10–2, 10–9
   sysIntEnable(), 10–2, 10–9
   sysLocalToBusAdrs(), 10–2, 10–10
new I/O routines
   sysVmeReadByte(), 10–2, 10–12
   sysVmeReadInt(), 10–3, 10–13
   sysVmeReadQuad(), 10–3, 10–14
   sysVmeReadWord(), 10–3, 10–15
   sysVmeWriteByte(), 10–2, 10–16

68K VMEbus interface
   new I/O routines (cont'd)
      sysVmeWriteInt(), 10–2, 10–17
      sysVmeWriteQuad(), 10–2, 10–18
      sysVmeWriteWord(), 10–2, 10–19
   porting example, 11–1
   probing the bus
      sysVmeMemProbe(), 10–3, 10–11

Lint command, 9–5, 9–6, 9–11
Loading a driver, 5–41
   by directly linking, 5–42
      with the application interface, 5–41
      with the shell, 5–41
loadLib library, 5–41
loadModule routine, 5–41
logLib library, 5–26

Makefile, 5–42
Makefiles, 5–42
malloc() routine, 5–26
mb() function, 3–4, 3–5
Memory barrier (MB) instruction, 3–4, 3–5, 9–3
memProbe() routine, 10–3

NCR 53C810 PS/20 SCSI driver, 8–34, 8–62
NCR 53C90 ASC SCSI driver, 8–34
netJobAdd() routine, 7–32
Nonblock devices
   See Device, nonblock
NSC PC87311/PC87312 floppy disk driver, 5–2
Outbound I/O
see Programmed I/O (PIO)

PCI bus, 2–3
Peripheral Component Interconnect (PCI) bus
See PCI bus
Pointer I/O, 5–24, 10–2, 10–3, 10–8
Porting, 1–7
See also Alpha architecture constants, 9–3
assignment guidelines, 9–4
data access, 9–2
data alignment, 9–3
data representation, 9–2
data examples, 11–1
file systems, 9–11
pointers, 9–10
assignment guidelines, 9–6, 9–10
structures, 9–7
alignment, 9–8
alignment guidelines, 9–8
size of, 9–7
truncation, 9–4, 9–7, 9–10
unavailable instructions, 9–11
using the lint command, 9–5, 9–6, 9–11
variables, 9–5
assignment guidelines, 9–5
used as pointers, 9–6
with the 68K VMEbus interface
See 68K VMEbus interface
Programmed I/O (PIO), 1–4, 4–3, 10–4

rawFS file system
See File systems
Read-modify-write operations, 5–40

rt11FS file system
See File systems

Scatter/gather mapping
see I/O addressing, scatter/gather mapping
scsiBlkDevCreate() routine, 8–5
scsiDebug global variable, 8–5
scsiIntsDebug global variable, 8–5
scsiLib library, 8–1, 8–3
scsiPhysDevCreate() routine, 8–5
SCSI versus nonSCSI nonblock devices, 8–3
SCSI_DEBUG_MSG() macro, 8–32
SCSI_INT_DEBUG_MSG() macro, 8–32
select() function, 4–4
implementing, 5–20
selectLib library, 5–21
selNodeAdd() routine, 5–21
selNodeDelete() routine, 5–21
selWakeup() routine, 5–21
selWakeupAll() routine, 5–21
selWakeupListInit() routine, 5–21
SelWakeupType() routine, 5–21
Semaphores
use in ISRs, 5–25
semLib library, 5–27
set_if_addr() routine, 7–33
Single-board computers (SBCs)
See VxWorks, target platforms
Sparse-space addressing
See I/O addressing, sparse space
splim() routine, 7–34
splnet() routine, 7–34
splx() routine, 7–35
stdio library, 1–5
Synchronization, 5–27
sysBusIntGen() routine, 10–2, 10–6
sysBusTas() routine, 10–2, 10–6
sysBusToLocalAdrs() routine, 10–2, 10–7
sysIntDisable() routine, 10–2, 10–9
sysIntEnable() routine, 10–2, 10–9
sysLib library, 8–3, 10–1
sysLocalToBusAdrs() routine, 10–2, 10–10
sysScsiInit() routine, 8–3, 8–4
sysVmeMemProbe() routine, 10–3, 10–11
sysVmeReadByte() routine, 10–2, 10–12
sysVmeReadInt() routine, 10–3, 10–13
sysVmeReadWord() routine, 10–3, 10–15
sysVmeWriteByte() routine, 10–2, 10–16
sysVmeWriteInt() routine, 10–2, 10–17
sysVmeWriteQuad() routine, 10–2, 10–18
sysVmeWriteWord() routine, 10–2, 10–19
sysWbFlush() routine, 9–3

T
tyCoDevCreate() driver routine, 6–7
tyCoDrv() driver routine, 6–9
tyCoInt() driver routine, 6–9
tyCoIoctl() driver routine, 6–10
ntyCoOpen() driver routine, 6–11
ntyCoStartup() driver routine, 6–12
tyLib library, 6–2

U
usrConfig.c configuration file, 5–43
usrRoot() routine, 5–31, 5–43
usrScsiConfig() routine, 8–4

V
VMEbus, 2–4
VME_READ_BYTE macro, 10–2, 10–12
VME_READ_INT macro, 10–3, 10–13
VME_READ_QUAD macro, 10–3, 10–14
VME_READ_WORD macro, 10–3, 10–15
VME_WRITE_BYTE macro, 10–2, 10–16
VME_WRITE_INT macro, 10–2, 10–17
VME_WRITE_QUAD macro, 10–2, 10–18
VME_WRITE_WORD macro, 10–2, 10–19
VxWorks
    hardware environment, 1–1
    I/O system

VxWorks
I/O system (cont’d)
    See I/O system
    kernel, 1–4
    software environment, 1–4
    target platforms
        AXPpci, 2–2
        AXPvme, 2–2
        EB64, 2–2
        EB64+, 2–3
        EB66, 2–3

W
Write buffer, 1–7, 3–2, 3–4
    write posting, 3–4

X
xxattach() routine, 7–12
xxBlkCreate() driver routine, 5–38, 5–44
xxBlkRd() driver routine, 5–11
xxBlkWrt() driver routine, 5–13
xxClose() driver routine, 5–8
xxCreat() driver routine, 5–6
xxDevCreate() driver routine, 5–32, 5–43
xxDrv() driver routine, 5–31, 5–43
xxInit() driver routine, 5–34, 5–44, 7–14
xxInt() routine, 7–15
xxIoctl() driver routine, 5–10, 5–15, 5–21, 7–16
xxOpen() driver routine, 5–7
xxOutput() routine, 7–17
xxRead() driver routine, 5–9
xxRemove() driver routine, 5–7
xxReset() driver routine, 5–18, 7–18
xxStatusChk() driver routine, 5–19
xxWatchdog() routine, 7–18
xxWrite() driver routine, 5–9
xxxBusPhaseGet() routine, 8–17
xxxBytesIn() routine, 8–18
xxxBytesOut() routine, 8–19
xxxCtrlCreate() routine, 8–20
xxxCtrlInit() routine, 8-22
xxxDevSelect() routine, 8-23
xxxDmaBytesIn() routine, 8-24
xxxDmaBytesOut() routine, 8-25
xxxFsDevInit() routine, 5-44
xxxIntr() routine, 8-26
xxxMsgInAck() routine, 8-27
xxxScsiBusReset() routine, 8-28
xxxScsiTransact() routine, 8-28
xxxSelTimeOutCvt() routine, 8-30
xxxShow() routine, 8-31
XYCOM 203 realtime clock driver, 4-5

Z
ZBC530 serial controller driver, 6-13