## Contents

1 Introduction .............................................................................................................. 1
  1.1 About This Document ....................................................................................... 1
  1.2 Additional Documentation .............................................................................. 2
  1.3 Terminology ...................................................................................................... 3

2 Design Overview .................................................................................................... 5
  2.1 Introduction ....................................................................................................... 5
  2.2 Design Goals .................................................................................................... 5
    2.2.1 Performance ............................................................................................. 6
    2.2.2 Code Flexibility and Portability ................................................................. 6
    2.2.3 Maintenance and Readability ................................................................. 7
    2.2.4 Ease of Configuration ............................................................................. 7
    2.2.5 Performance Testing .............................................................................. 7
    2.2.6 Code Size ................................................................................................ 8
    2.2.7 Reentrancy and Number of Devices ...................................................... 8
  2.3 Common Design Concerns ............................................................................... 8
2.3.1 Varied Hardware Designs ................................................................. 9
2.3.2 Memory-Mapped Chips ................................................................. 9
2.3.3 I/O-Mapped Chips ..................................................................... 9
2.3.4 Multi-Function Chips ................................................................. 10
2.3.5 Bus Support ................................................................. 10
   VxBus-Compliant Drivers ......................................................... 11
   Legacy Drivers .................................................................. 11
   VxBus Bus Controller Device Drivers .................................. 12
2.3.6 Interrupt Controllers ............................................................... 12
2.4 Design Guidelines ................................................................. 12
   2.4.1 Names and Locations ...................................................... 13
   2.4.2 Documentation and Standards ........................................ 13
   2.4.3 Per-Device Data Structure ............................................ 15
   2.4.4 Per-Driver Data Structure ............................................. 15
   2.4.5 Driver Interrupt Service Routines .................................. 15
   2.4.6 Access Macros ............................................................. 16
   2.4.7 Multi-Function Devices and VxBus .................................. 18
3 Adding Drivers to VxWorks ................................................................. 19
   3.1 Adding an Existing Driver to Your BSP ..................................... 19
      3.1.1 BSP Support for Legacy (Non-VxBus) Device Drivers .... 20
      3.1.2 Project Facility ............................................................. 20
      3.1.3 Component Descriptor Files ....................................... 21
   3.2 Porting Drivers ................................................................. 21
      3.2.1 Porting a Driver From An Earlier Version of VxWorks .... 21
      3.2.2 Porting a Legacy Driver to the VxBus Model ............... 21
      3.2.3 Porting From Another OS ........................................... 22
   3.3 Creating a New Driver Using the Legacy Model ..................... 23
3.3.1 Step-By-Step Instructions ................................................................. 23
   Top-Down Design ........................................................................ 23
   Bottom-Up Implementation ......................................................... 25
   Helpful Hints ........................................................................... 26

3.3.2 Cache Considerations ................................................................. 26
   Using the Cache Library ............................................................. 27
   Developing a cacheLib Strategy .................................................. 31

3.3.3 Hotswap Support ......................................................................... 35

3.3.4 Downloadable Driver Support .................................................... 35

4 END Ethernet Drivers ........................................................................ 37

4.1 Introduction ..................................................................................... 37

4.2 END Driver Overview ...................................................................... 38
   4.2.1 Driver Environment ............................................................. 38
     The MUX ............................................................................... 38
     Network Interface Drivers and Protocols .................................. 39
     The MUX, Protocol, and Driver API ........................................ 40
     Driver Components ................................................................ 42
     Protocols That Use the MUX API .......................................... 43
     Interactions With the MUX API ............................................. 47
     Network Layer to Data Link Layer Address Resolution ........... 53
   4.2.2 VxWorks OS Interface ........................................................... 54
     Understanding How VxWorks Launches and Uses Your Driver ... 54
     Executing Calls Waiting In the Network Job Queue ............... 58
     Adding Your Network Interface Driver to VxWorks ................ 59
     Allocating, Initializing, and Utilizing Memory Resources .......... 61
     Handling Packet Reception .................................................... 69
     Handling Packet Transmission .............................................. 82
     Implementing Checksum Offloading ........................................ 88
     Implementing Required Entry Points and Structures ............... 95

4.3 The END Driver Development Process ............................................ 118
4.3.1 Driver Development Overview .............................................................. 118
  Writing a New Driver .............................................................................. 119
  Porting an Existing Driver From Another OS ..................................... 120
  Additional Development Issues ............................................................. 121
4.3.2 Error Conditions ...................................................................................... 122
4.3.3 Generic MIB Interface Initialization ...................................................... 123

5  BSD Ethernet Drivers ........................................................................... 131
  5.1 Introduction ................................................................................................. 131
  5.2 BSD Ethernet Driver Overview ................................................................. 132
    5.2.1 Structure of a 4.3 BSD Network Driver ................................................. 132
    5.2.2 Etherhook Routines Provide Access to Raw Packets ......................... 134
  5.3 Upgrading Your Driver to 4.4 BSD ............................................................ 135
    5.3.1 Removing the xxOutput( ) Routine ....................................................... 136
    5.3.2 Changing the Transmit Startup Routine ............................................... 136
    5.3.3 Adapting to Changes In Receiving Packets .......................................... 137
    5.3.4 Creating a Transmit Startup Routine ..................................................... 137
  5.4 Porting a BSD Ethernet Driver to the END Model ...................................... 139
    5.4.1 Rewriting xxattach( ) to Use an endLoad( ) Interface ......................... 139
    5.4.2 Handling Task-Level Packet Reception with xxReceive( ) ................. 140
    5.4.3 Rewriting xxOutput( ) to Use an endSend( ) Interface ....................... 140
    5.4.4 Using xxIoctl( ) as a Basis for endioctl( ) ............................................. 140

6 Flash File System Support with TrueFFS .............................................. 141
  6.1 Introduction .................................................................................................. 141
  6.2 TrueFFS Overview ....................................................................................... 142
    6.2.1 Core Layer ............................................................................................. 142
## Contents

6.2.2 MTD Layer ................................................................. 142
6.2.3 Socket Layer .............................................................. 143
6.2.4 Flash Translation Layer ............................................... 143

6.3 The TrueFFS Driver Development Process ..................... 143
6.3.1 Using MTD-Supported Flash Devices ...................................... 143
   Supporting the Common Flash Interface (CFI) .............................. 144
   Supporting Other MTDs .............................................................. 146
   Obtaining Disk-On-Chip Support .............................................. 148
6.3.2 Writing MTD Components .................................................. 148
   Writing the MTD Identification Routine ..................................... 148
   Writing the MTD Map Routine ................................................... 152
   Writing the MTD Read, Write, and Erase Routines ..................... 153
   Defining Your MTD as a Component ........................................ 155
   Registering the Identification Routine ....................................... 156
6.3.3 Socket Drivers ........................................................... 157
   Porting the Socket Driver Stub File ........................................... 158
   Understanding Socket Driver Functionality ................................ 162
6.3.4 Flash Translation Layer .................................................. 166
   Terminology ............................................................................. 166
   Overview ................................................................................... 171
   Structures .................................................................................. 171
   Erase Units ............................................................................... 178

7 SCSI Drivers ................................................................. 185
7.1 Introduction ................................................................. 185
7.2 SCSI Overview ............................................................ 186
   7.2.1 Layout of SCSI Modules ...................................................... 186
   7.2.2 The VxWorks OS Interface ................................................... 190
     Libraries .................................................................................. 190
     Driver Programming Interface ................................................. 193
7.3 SCSI BSP Interface ........................................................ 224
### 7.4 The SCSI Driver Development Process ......................................................... 226

### 7.5 Common SCSI Driver Development Issues .................................................. 227

#### 7.5.1 Troubleshooting and Debugging ......................................................... 227

#### 7.5.2 Test Suites .......................................................................................... 228

- scsiDiskThruputTest() ............................................................................. 228
- scsiDiskTest() ....................................................................................... 229
- scsiSpeedTest() ..................................................................................... 230
- tapeFsTest() .......................................................................................... 231

### 8 Timestamp Drivers ...................................................................................... 233

#### 8.1 Introduction .............................................................................................. 233

#### 8.2 Timestamp Driver Overview ..................................................................... 234

##### 8.2.1 Hardware Environment ................................................................. 234

##### 8.2.2 VxWorks OS Interface ..................................................................... 238

- Working with the Wind River System Viewer ........................................... 238
- Timestamp Driver Components ................................................................. 239
- Sample Drivers .......................................................................................... 240

#### 8.3 Timestamp Driver Configuration and BSP Interface ............................... 254

- sysTimestampConnect( ) ......................................................................... 255
- sysTimestampEnable( ) ............................................................................ 256
- sysTimestampDisable( ) ........................................................................... 256
- sysTimestampPeriod( ) ............................................................................ 256
- sysTimestampFreq( ) ................................................................................ 256
- sysTimestamp( ) ..................................................................................... 257
- sysTimestampLock( ) ............................................................................... 257

#### 8.4 The Timestamp Driver Development Process .......................................... 258

##### 8.4.1 Timers that Can Be Read While Enabled .......................................... 258

- Timer Period ........................................................................................... 258
- Interrupt Level ......................................................................................... 258
- Interrupt Locking ..................................................................................... 258
8.4.2 Working Around Deficiencies In Hardware Timers ................. 258
  Timer Re-Synchronization ................................................................. 259
  Timer Period .................................................................................. 259
  Down Counter ................................................................................ 259
  Counter Preloading ........................................................................ 260
  Adjustment for Time Skew ............................................................. 260
  Counter Read Optimization .............................................................. 260

8.4.3 Using the VxWorks System Clock Timer ................................. 260
  Timer Rollover Interrupt ................................................................. 261
  Timer Counter Not Reset ............................................................... 261
  Timer Period ................................................................................ 261

8.5 Common Timestamp Driver Development Issues ...................... 262

9 Additional Drivers ........................................................................ 263
  9.1 Introduction .................................................................................. 263
  9.2 ATAPI Drivers .............................................................................. 263

9.3 Interrupt Controller Drivers ...................................................... 264
  9.3.1 VxWorks 6.x ............................................................................ 264
  9.3.2 VxWorks 5.5 ............................................................................ 264
  BSP Interface .................................................................................... 265
  Non-Vectored Interrupt Sources .................................................... 266

9.4 Memory Drivers .......................................................................... 266
  9.4.1 Hardware Mismatches ............................................................. 267
  9.4.2 Complex Modern Memory Controllers .................................. 267

9.5 Multi-Mode (SIO) Serial Drivers .............................................. 268
  9.5.1 SIO_CHAN and SIO_DRV_FUNCS ........................................... 269
  9.5.2 Polled Mode, WDB, and Kernel Initialization ......................... 271
  9.5.3 Serial Ports, WDB, and Interrupts ......................................... 271

9.6 Serial Drivers ............................................................................. 272
10 VxBus ................................................................. 275

10.1 About VxBus .................................................. 275

VxBus Hierarchy ................................................. 276

10.2 Device Models .................................................. 277

10.3 Terminology ................................................ 278

10.4 Bus Controller Devices .................................. 280

10.5 Services ......................................................... 280

10.5.1 Device Discovery ......................................... 280

10.5.2 VxBus Interfaces ......................................... 281

10.5.3 Initialization Sequence ................................. 282

10.5.4 Bus Topology ........................................... 282

10.6 Multi-Stage Initialization ................................ 283

First Stage Initialization ................................... 284

Second Stage Initialization ................................. 284

Third Stage Initialization ................................... 285

10.6.1 OS- and Middleware-Specific Initialization .... 285

10.6.2 Device Probe Routine ................................ 285

10.7 Driver Methods .............................................. 288

10.7.1 OS and Middleware Access to Driver Methods 288

Classes, Types, and Macros ............................... 290

10.8 Register Access ............................................. 291

10.9 Interrupts ....................................................... 292

10.9.1 VxBus Interrupt Management ....................... 292

Interrupt Control Routine API Reference ............... 293
11 VxBus Device Drivers ................................................................. 297

11.1 Introduction .................................................................................. 297

11.2 VxBus Registration and Initialization ........................................... 298

11.3 Driver Registration and Initialization ............................................ 298
   11.3.1 Device Driver Registration .................................................... 299
   11.3.2 PLB Registration ............................................................... 300
   11.3.3 PCI Registration ............................................................... 300

11.4 Device Access in the Register Device Model ............................... 301
   11.4.1 pAccess Structure ........................................................... 301
   11.4.2 Standard Device Registers ................................................. 302
      Access Routines Provided by VxBus ........................................... 304
      Optimized Access Routines Provided by the BSP ........................... 304
      Optimized Access Routines Provided by the BSP and Multiple Bus
      Locations ............................................................................... 305
   11.4.3 Configuration Space Registers .......................................... 305
   11.4.4 Volatile Registers ............................................................ 306
   11.4.5 Register Probe Routine ..................................................... 308

11.5 Interrupts .................................................................................. 308

11.6 Driver Methods ........................................................................... 310

11.7 Multi-Function Devices and VxBus Virtual Buses ....................... 310

11.8 Common Issues .......................................................................... 311

11.9 Debugging Your Device Driver .................................................. 312
   11.9.1 VxBus Show Routines ....................................................... 312
   11.9.2 VxBus Debug Messages ................................................... 313

11.10 Integrating Your Driver Code with VxBus ................................. 313
12 VxBus Bus Controller Device Drivers ....................................................... 315
  12.1 Introduction ................................................................................................ 315
  12.2 Initialization and Registration ................................................................. 315
13 VxBus Methods and Utilities ...................................................................... 317
  13.1 Introduction ................................................................................................ 317
  13.2 VxBus Parameter System ............................................................................ 317
      13.2.1 VxBus Parameter System Functions .................................................. 318
      13.2.2 Using the Parameter System .............................................................. 319
          Data Structures .......................................................................................... 319
      13.2.3 Parameter System Interfaces ............................................................... 320
  13.3 Timer Abstraction Routines ....................................................................... 321
14 Hierarchical END Drivers (hEND) ............................................................ 323
  14.1 Introduction ................................................................................................ 323
  14.2 Architecture Overview ............................................................................. 325
      14.2.1 Command and Control ....................................................................... 327
      14.2.2 Receiver .............................................................................................. 327
      14.2.3 Transmitter ......................................................................................... 327
      14.2.4 VxBus Hardware Access Abstraction .................................................. 328
  14.3 Driver Files ................................................................................................ 328
      14.3.1 File Integration .................................................................................... 329
  14.4 Driver Configuration ................................................................................... 329
      14.4.1 Configuration with CDF ....................................................................... 330
      14.4.2 Configuration in hwconf.c ................................................................. 330
      14.4.3 Default Configuration ......................................................................... 330
14.4.4 Configuration from an Application ........................................... 331
    Interrupt Configuration ........................................................... 331
14.4.5 Configuring a Task Level Routine to Run in a Non-default Context 332

14.5 Loading an hEnd Driver .......................................................... 332

14.6 Unloading an hEnd Driver ....................................................... 333

14.7 Developing a New Driver ....................................................... 333
    14.7.1 The Development Process .............................................. 334
    14.7.2 Additional Tasks .......................................................... 336
        Optimized Access Macros .................................................. 336
        Command-Line Build Capability ..................................... 336
        Parameter System Integration ........................................ 337

Index ........................................................................................................ 339
1.1 About This Document

This document presents information intended for VxWorks device-driver developers. This includes developers who are responsible for:

- creating new device drivers from scratch
- upgrading existing device drivers for the latest hardware revision
- porting device drivers from other operating systems
- maintaining existing device drivers

This document includes both generic instructions for approaching the VxWorks device-driver development process and detailed information about many specific VxWorks device-driver types such as END drivers, TrueFFS drivers, and timestamp drivers. The later chapters of this document describe the VxBus infrastructure (available in recent VxWorks 6.x releases) and provide information on implementing VxBus device drivers. For more information on VxBus, see 10. VxBus.
Specific device-driver information is also available from the template drivers provided in your installation. Template drivers are available for most driver types and are located in the `installDir/vxworks-6.x/target/src/drv/TYPE` directory.

This document also includes a general discussion of cache considerations for DMA driver design. However, you should refer to the appropriate *VxWorks Architecture Supplement* document for architecture-specific details of cache implementations.

**NOTE:** In general, this document applies to *VxWorks* 5.5 users as well as *VxWorks* 6.x users. Features or requirements that apply to a specific version of *VxWorks* are marked as necessary. *VxWorks* 5.5 users are strongly encouraged to review the *VxWorks BSP Developer’s Guide, 5.5* for additional information.

### 1.2 Additional Documentation

Before beginning any device-driver development, you should have a good understanding of the overall *VxWorks* I/O system. For more information, see the *VxWorks Kernel Programmer’s Guide*.

In addition, you may want to reference the following *VxWorks* companion documents:

- *VxWorks BSP Developer’s Guide*—This document discusses *VxWorks* BSP development. In particular, it provides guidelines for writing a custom BSP based on an existing reference BSP.

- *VxWorks Hardware Considerations Guide*—This document discusses issues related to embedded hardware design with a focus on *VxWorks*. It provides guidelines and suggestions for selecting hardware for a *VxWorks*-based project.
1.3 Terminology

The following terminology is used in this document:

installDir
Within this document, file paths are typically expressed as a full path; this practice maintains consistency between this and other Wind River documentation. Although this document is intended for use with both VxWorks 5.x and VxWorks 6.x, the standard for specifying paths is the VxWorks 6.x directory structure, for example:

```
installDir/vxworks-6.x/target/src/drv/scsi/Makefile
```

In VxWorks 5.x the vxworks-6.x level did not exist, so the corresponding VxWorks 5.x path would be:

```
installDir/target/src/drv/scsi/Makefile
```

bspname
In several places within this document, there are references to filenames that are based on the BSP. These filenames have the string bsiname substituted. For example, if you are working on a BSP called acmeBSP, change any reference bsiname to acmeBSP. For example, bsiname.h would become acmeBSP.h.

type
Drivers for specific devices are groups by device type. For example, SCSI drivers are located at installDir/vxworks-6.x/target/src/drv/scsi. For the general case, type represents the device type:

```
installDir/vxworks-6.x/target/src/drv/type.
```

dev
Where this document refers to devices in general, these devices are generically referred to as dev. In such cases, substitute the name of each device or device type for dev. For example, if your driver supports ncr810, the general file devinit.c becomes ncr810init.c.
2.1 Introduction

This chapter provides an overview of driver design considerations. These general concepts are discussed in specific detail in following chapters.

2.2 Design Goals

VxWorks is an operating system for real-time and embedded applications. This places some constraints on the design of device drivers.

The primary goal for most VxWorks drivers is real-time performance of the target system as a whole. In general, if a driver does not allow real-time execution of
applications running on the target, the driver is a poor choice for use with VxWorks and another driver should be selected. Depending on the application, this may be an absolute requirement, or it may be an important consideration.

Memory footprint is another constraint for VxWorks drivers. Many embedded applications have limited memory and because demand paging to disk is not compatible with real-time operation, memory constraints are extremely important.

Standard software requirements are also important in the VxWorks environment. This includes requirements such as driver flexibility, code maintainability, code readability, and driver configurability.

### 2.2.1 Performance

Drivers must perform well enough to match the real-time kernel's abilities. Designing for performance implies many things. First, it requires using direct memory access (DMA) and interrupts in an efficient manner. This requires you to keep your routine nesting at an optimum level. For example, too many routine calls and restore operations can increase process dispatch latency and reduce performance. However, performance requirements must be balanced against proper use of routines for keeping code size small and making your driver design easy to follow and understand.

Designing for performance also means keeping interrupt latency to a minimum. Interrupt handlers must receive the greatest care in any design. Overall system performance is just as important as the specific driver's performance.

For specific applications, you may consider it acceptable to write a VxWorks driver that sacrifices one or more of these goals. For example, when writing a driver for a system that is expected to be used only for a specific non-real-time application, you may be tempted to sacrifice real-time system performance in your driver design. However, because of issues such as code re-use, Wind River strongly discourages this approach. Real-time performance and memory footprint are an important concern for all VxWorks drivers.

### 2.2.2 Code Flexibility and Portability

In a flexible design, your device driver must adapt to new board configurations with minimal effort. Key elements of this design consideration include structured design features and use of macros for all hardware accesses. Flexibility comes in two forms: run-time flexibility and compile-time flexibility. Run-time flexibility typically sacrifices some amount of real-time performance for an object module.
that uses pointers to access routines. This generally achieves the desired flexibility. (Run-time flexibility is also referred to as portability.) Compile-time flexibility employs the use of preprocessor macros to customize the system at compile-time for performance.

Wind River recommends using both methods whenever possible. This gives compiled object modules the desired portability and still allows the same source code to be compiled with a different set of macros to generate an optimized module. The preferred method at Wind River is to use compile time macros to implement run-time vectored routines. This achieves both goals. A compiled object module can be customized at run time to achieve flexibility. Yet, that same source code can be used with redefined compile time macros to create a module optimized for performance.

2.2.3 Maintenance and Readability

Most of the effort involved in software engineering is maintenance. Therefore, any effort that reduces the maintenance burden is valuable. By adhering to coding standards and producing quality documentation, you make your code easy to read, easy to understand, and easy to maintain. Poor quality documentation is just as detrimental to the maintenance process as insufficient documentation. Any new device driver documentation should be reviewed by at least one objective person (not the author of the code).

2.2.4 Ease of Configuration

Your driver should not limit the end user’s options or requirements. Do not impose limits on the number of devices that can be supported or on other features. You may not be able to support all device features or operating modes in your original driver, but your design should not preclude expanded device support at a later time.

2.2.5 Performance Testing

All drivers must be tested for expected behavior, and all drivers should be tested for performance. In addition to writing the driver functionality, you must also consider writing test routines. This involves inserting debug information into your code as well as supporting benchmark tests. If a standard benchmark test is not available, you must consider writing one. You should consider testing for both
performance and expected behavior regardless of your driver type (Ethernet, SCSI, serial, timers, interrupt controllers, and so forth).

In general, high-level debug code such as that used during performance testing should be well-written, surrounded by `#ifdef/#endif` statements, and left in the source code in order to ease future debugging efforts.

### 2.2.6 Code Size

In the embedded real-time operating system (RTOS) market, code size (footprint) is important. Code size should be minimized through structured design. However, reducing code size can hurt performance. As a developer, you must balance your design such that you provide adequate performance without excessive code size.

### 2.2.7 Reentrancy and Number of Devices

Drivers should be fully reentrant in order to support any number of devices. Access to global data structures must be protected by some form of synchronization. The usual synchronization methods include `intLock()`, `taskLock()`, or a mutex semaphore.

Drivers that limit the number of supported devices are not desirable. Instead of using fixed arrays of device information, you should create a structure for each new device and pass it to the driver for initialization and control. Alternatively, the driver can call the `malloc()` routine for the structure of each device as part of the device initialization call.

For example:

```c
#define NUM_LN 2
LN_CTRL ln_softc[NUM_LN]; /* BAD IDEA */
```

### 2.3 Common Design Concerns

This section discusses the design concerns associated with a variety of hardware designs, memory-mapped chips, I/O-mapped chips, multi-function chips, multiple buses, and interrupt controllers.
2.3.1 Varied Hardware Designs

The variety of hardware designs available today is nearly unlimited. You cannot assume anything about your target system hardware. Hardware designers continue to push the limits of design, requiring software to play an increasing role in the overall system.

Ideally, your first encounter with a new chip tells you most of what you need to know for all subsequently derived chips. However, in most cases, experience differs sharply from this ideal. Often, the first hardware implementation, and thus the first software driver, are sure to require extensive modification by the time the hardware technology matures.

2.3.2 Memory-Mapped Chips

In memory-mapped systems, hardware designers make choices about how to map the chip registers into the memory space. Systems using the same chip can map the device I/O registers in very different ways.

Consider a simple serial chip with four registers, each with a width of one byte. One designer might map the registers to four consecutive byte addresses. Another designer, using a 16-bit memory system, might map each register on a half-word boundary using only the low order 8 bits of the 16-bit memory bus. In another system, the registers might be mapped to long word addresses. These are three possible implementations with three different addressing schemes.

You should also consider that the designer who chose the long word implementation might also require the driver to use only long word read/write accesses. Therefore, you cannot assume that a byte read operation can be used to read a byte register.

One solution to this problem is to require that all accesses to the chip be restricted to as few routines as possible. Further, each access to the chip should be declared in a preprocessor macro that can be redefined in a way that meets the special requirements of any particular system.

2.3.3 I/O-Mapped Chips

Many device driver developers have their first hardware experiences with processors that use memory mapped I/O exclusively. However, there is another class of processors that use a separate address space, referred to as I/O space. The C language does not provide a means to specify if an address is a memory address
or an I/O address. Instead, you must use assembly language routines to access the special instructions that can reach data in I/O space. You cannot write a direct expression in C for this purpose.

The solution to the memory-mapped problem described in 2.3.2 Memory-Mapped Chips, p.9 solves this problem as well. By defining the special hardware access macros to use a special I/O routine, you can handle devices mapped into I/O space.

2.3.4 Multi-Function Chips

The trend in hardware design has been to combine more and more channels (devices) per chip. This led to the development of ASIC chips that combine multiple devices of different types into a single piece of silicon. Currently, designers are only limited by imagination in the ways they can combine silicon building blocks.

A single large monolithic driver for an entire ASIC is opposed to the goal of system scalability. The user should be able to exclude features that are not needed by the application. Therefore, rather than write a driver for a complete ASIC, you should strive to write drivers as though each subsection is a separate device. It might be necessary, though undesirable, for one driver to require support from another driver. If this is the case, this must be clearly documented in the dependent driver documentation.

2.3.5 Bus Support

There are several bus types available on target systems, including processor local bus (PLB), VMEbus, VXI, QBus, SBus, PCI bus, RapidIO bus, and so forth. Embedded systems can now include the same level of complexity as non-embedded computer systems. Some buses actually define their own byte ordering, which can be different from the local CPU byte ordering.

One example of a complex board system is the Motorola MVME-1600 board. The local bus is the PowerPC MPC bus. The MPC105 chip connects the MPC bus to a PCI bus. A PCI chip connects to an ISA Bus. A different PCI chip connects the PCI bus to a Motorola 68040 bus. From the 68040 bus, there is a VME chip to interface to the VMEbus. The VMEbus supports seven interrupt request levels and 256 possible vectors. The PCI bus supports four interrupt levels. The ISA bus supports 16 interrupt levels. PCI is little-endian by definition. The 68040 bus is big-endian by definition. The MPC bus is ambidextrous (either endian).
Even with a complex bus design on your hardware, your goal should be to have drivers that work on any bus with a minimum number of configuration changes.

**VxBus-Compliant Drivers**

When working with the VxWorks VxBus infrastructure (see 10. VxBus), maintaining portability is relatively easy. The facilities provided by VxBus, such as register access routines, handle any changes that are required in order for the driver to talk with the device. This includes byte-order transformations, write posting, processor pipeline flush, and other operations.

The remaining portability issues that VxBus device driver developers need to consider are mostly related to the processor. You must ensure that reads and writes are performed at appropriate boundaries and that the base address plus an offset is used to refer to registers, rather than referring to registers through a structure pointer that could be modified by a compiler. In addition, you must provide protection for device descriptors and other memory objects that are manipulated by both the device and the processor.

For more information about the facilities provided by VxBus, see 10.1 About VxBus, p.275.

**Legacy Drivers**

Legacy (or non-VxBus) VxWorks device drivers are split into two source files. The core driver resides in a directory separate from the BSP, and should be written to be as flexible and portable as possible. The BSP support file, `sysDev.c`, resides in the BSP and contains code which is specific to the BSP.

The core driver should use a set of macros to perform accesses to the device registers. These macros are available for the BSP developer to fill in; they contain any required transformations such as byte-order manipulation, processor pipeline flush, or other operations.

If you are creating and supporting legacy drivers, you must also handle read and write boundary conditions, register address calculations, and protection of descriptors, as described in VxBus-Compliant Drivers, p.11.
VxBus Bus Controller Device Drivers

Unlike in legacy systems, the software that manages the bus controller device (or devices) is a VxBus device driver. This means that bus controller software can now be reused more easily, without copying the source file into a new BSP directory.

2.3.6 Interrupt Controllers

Some hardware designs require special interrupt acknowledge steps. Sometimes interrupts are processed through cascaded chains of interrupt controllers, requiring multiple steps with each interrupt, just to reset or clear the interrupt controller hardware.

You must ensure that all of your interrupt service routines are compatible with chaining. This means that your driver code must determine if the device is actually asserting an interrupt request and if the device is not asserting a request, the code must exit immediately.

Ideally, the device driver is not concerned with connecting interrupt vectors. This should be the responsibility of the board support package (BSP). The driver interrupt service routine (ISR) should be a global routine that the BSP connects to the vector as part of `sysHwInit2()`. If a driver must be involved in the interrupt connect step, the connection should be implemented through a hardware abstraction macro. This needs to be a flexible interface so that different boards with different interrupt structures can be supported from a single driver.

2.4 Design Guidelines

This section discusses the guidelines you should follow when designing a driver for use with a Wind River product. Included are discussions of routine naming conventions, documentation standards, per-device and per-driver data structures, interrupt service routines, and access macros for chip registers.
2.4.1 Names and Locations

In general, you should follow Wind River driver naming conventions when naming your driver. You must be careful when naming routines and variables. Each module should have a distinct prefix and every routine and variable that you declare must start with that prefix. Wind River coding conventions suggest the module-noun-verb method of name construction. You can also consider the naming convention as proceeding from generic to specific going from left to right. Related routines must have a common root and be distinguishable from one another by their suffixes.

A poor example:

```c
STATUS fooStartPoll (void);
STATUS fooStopPoll (void);
```

A better example would be (assuming both routines are related to a polling operation):

```c
STATUS fooPollStart (void);
STATUS fooPollStop (void);
```

Only Wind River generic drivers are stored in `installDir/vxworks-6.x/target/src/drv/xxx`. This implies that all drivers in this directory are used by more than one BSP, or are expected to be used by more than one BSP. Third party BSP writers frequently assume that these directories are for all drivers, including their own. Because the future is unpredictable, Wind River reserves all rights with respect to these directories. Third parties are encouraged to create a `installDir/vxworks-6.x/target/src/drv/mfg` directory and a `installDir/vxworks-6.x/target/h/drv/mfg` directory and put driver source files and header files in those directories. For example, Mom and Pop’s Software Shoppe might create `installDir/vxworks-6.x/target/src/drv/momandpop` and `installDir/vxworks-6.x/target/h/drv/momandpop`. Alternatively, you may place your drivers in the same directory as your BSP. In some cases, Wind River special drivers are placed in the BSP directory.

Occasionally, a custom version of a driver is simply a file that contains a set of wrappers that declare a specialized set of macros and include the generic driver (using `#include`). Customized drivers of this nature should always be put in the BSP directory.

2.4.2 Documentation and Standards

Standard documentation includes a comment block at the beginning of the driver source file that contains documentation for the driver. This documentation should
describe the entire chip with all of its supported features and operating modes. Developers responsible for code maintenance should be able to determine the basis of the chip without resorting to a complete library search. Most manufacturers now post their data sheets on the World Wide Web. This makes finding data sheets easier. However, a well-written one paragraph summary of a chip can save a lot of time spent in searching and reading.

Your driver documentation should also describe the driver modes and limitations. The module description should be an introduction to both the chip and the driver.

Documentation should follow the Wind River coding standard (see VxWorks BSP Developer's Guide: Documentation Conventions). This enhances readability and eases maintenance. If possible, you should have a technical writer or editor review the documentation for clarity and proper grammar.

You should also perform a formal code review for your driver to check for adherence to coding standards. A code review ensures the quality and readability of your driver is at an acceptable level.

The documentation should focus on how the device and the driver work together. In addition to missing documentation, device drivers often have trivial or redundant documentation. For example:

```
x = 0;    /* clear x */
fooReset (&myFoo); /* reset myFoo */
```

Assembly-level programmers document each line of code. In C, programmers tend to write comments preceding the code. The line-by-line comment model often generates insufficient documentation for device drivers. Wind River recommends that you avoid line-by-line running comments in your device driver.

The documentation that goes into generated documentation is generally more important (and visible) than comments in the body of the code. The generated documentation is accessed by the user therefore the documentation must be accurate and clear. When documented properly, the reader should understand what the code intends to accomplish. When the user clearly understands the code, it is much easier to identify problems.

Your documentation should also describe how to integrate the driver into an overall BSP package. The documentation should include all special external routines as well as instructions for initializing the driver. You might also consider supplying examples of BSP integration code.
2.4.3 Per-Device Data Structure

As part of an object-oriented design, each device should be represented in the system by a single structure that includes all state information. The object methods (routines) perform operations on objects using an object handle (a pointer to the data structure representing the object).

You should create new instances of a device object by calling a device create routine in the driver library (xxx\texttt{DevCreate()}). At device creation time, the driver should verify that the device is actually present. If the device is not present, the device creation operation should fail and return a null pointer.

2.4.4 Per-Driver Data Structure

In keeping with object-oriented design methodology, you should also create a structure to represent the driver itself. This structure should include all driver state information (class variables). If you have all driver data in a single structure, it is easy to display data using a debugging tool.

2.4.5 Driver Interrupt Service Routines

Because there may be bus issues related to interrupts, drivers should not call \texttt{intConnect()} directly. Your driver should define a macro that can be changed by the user to call the correct interrupt connection routine (which may not be \texttt{intConnect()}). For example:

\begin{verbatim}
#ifndef FOO_INT_CONNECT
#define FOO_INT_CONNECT(vec, rtn, arg) intConnect(vec, rtn, arg)
#endif
\end{verbatim}

If the device is not asserting an interrupt, the device driver ISRs must exit immediately. Do not assume that there is a one-to-one mapping between interrupt vectors and interrupt handlers. With PCI systems, it is likely that interrupt lines are used to service more than one device. Interrupt routines must examine the device and determine if it is actually generating an interrupt condition. If the device is not generating an interrupt, your interrupt handling code should exit as quickly as possible.
2.4.6 Access Macros

Every access to chip registers should be made through macros that can be redefined to accommodate different access methods. Usually, a read and a write macro are all that is required. In some cases, you must also supply a modify macro to change individual bits in a register. For example:

\[
\begin{align*}
M68681\_READ & (addr, pData) \\
M68681\_WRITE & (addr, data) \\
M68681\_CLR\_SET & (addr, clear\_bits, set\_bits)
\end{align*}
\]

Accesses to a command block in memory, or on a bus, must also be made through a macro that can be redefined to accommodate byte swapping and address manipulation.

Note that read macros are passed the address of where to return the value that is read. These macros do not return a value in the same manner a routine returns a value. This is deliberate. Macros written this way can be replaced by simple statements, routine calls, or by a block statement. Macros written to return a value cannot be replaced by a C block statement. For example:

\[
\begin{align*}
xxx = M68681\_READ & (x); /* Limited */ \\
M68681\_READ & (x, &xxx); /* Better */
\end{align*}
\]

You should strive to minimize preprocessor conditional expressions within code blocks. These conditional expressions should add to the readability of the code, not reduce it. If the conditional only changes the value of an argument to a routine, it should be done at the beginning of the file using a \#define. Only conditional expressions that actually change the flow of logic should be within a routine.

In the following example, the only change is to one argument of a routine call. Putting the conditional statements inside the routine confuses the read because the conditionals have absolutely no effect on the flow of execution.

\[
\begin{align*}
\#ifdef \ INCL\_FOO \\
\text{fooReset} & (\&myFoo, \\
\text{else} \\
\text{fooReset} & (\&yourFoo, \\
\text{endif} \\
\text{arg2, arg3, arg4});
\end{align*}
\]

To fix this situation, you must create a new macro that is the value of the argument. Define the macro at the head of the file, then refer to it at the proper place in the code. This results in a routine that is much easier to understand. For example:
In general, you should use conditional compilation only if the conditional changes the flow of execution within the routine. If the changes are limited to argument values, variable names, or routine names, define an intermediate macro as described previously.

You must use caution when creating macro names. The names should be neither too simple nor too complex. Both of the following examples are unsatisfactory:

```c
SIO_READ
NS16550_CONTROL_REGISTER_BIT_7_CLEAR
```

Do not use structures to declare hardware registers or memory control blocks. You must use macros for offset constants or to convert a base address into a register address. Structures are inherently non-portable. The use of structures is a common source of failure when porting drivers across architectures. Even using the same toolchain, the default structure creation can vary from architecture to architecture. The C language specification does not specify a standard for structure element alignments.

```c
/* not portable */
typedef struct
{ /* I8250_DEV */
    char CSR;
    char DATA;
    char MBR;
} I8250_DEV;
/* A better way */
#define I8250_DEV <some address>
#define I8250_CSR 0
#define I8250_DATA 1
#define I8250_MBR 2

#define I8250_READ(offset, value) \value = *(I8250_DEV + offset);
#define I8250_WRITE(offset, value) \*(I8250_DEV + offset) = value;
```
2.4.7 Multi-Function Devices and VxBus

Wind River recommends that drivers for multi-function devices be divided into separate drivers for each of the individual sub-components of the multi-function device. This allows drivers for unused parts of the device to be scaled out of the system to decrease memory requirements and also improves code reusability for device drivers.

However, at times it can be more convenient to manage drivers by including a single unit during configuration of a VxWorks system. This can be accomplished simply with the VxBus design provided the drivers for the individual components of the device are well written. For more information about using VxBus to represent multi-function devices, see 11.7 Multi-Function Devices and VxBus Virtual Buses, p.310.
3.1 Adding an Existing Driver to Your BSP

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

- Providing BSP support.
- Integrating the driver with the appropriate configuration facilities (GUI and command line).
- Providing appropriate component description file (CDF) entries. (For information on CDF files, see the VxWorks Kernel Programmer’s Guide: Kernel.)
3.1.1 **BSP Support for Legacy (Non-VxBus) Device Drivers**

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

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

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

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

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

3.1.2 **Project Facility**

For VxWorks 5.5, integration with the project facility may involve creation of project configlettes in the directory `installDir/vxworks-6.x/target/config/comps/src`. The naming convention of these files should be `usrDev.c`. For example, the configlette for serial drivers is `usrSerial.c`.

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

For the driver to be selectable in the Wind River development suite environment (Workbench), there must be an entry for it in a CDF file and this entry must be brought into the project facility folder hierarchy. CDF files reside in `installDir/vxworks-6.x/target/config/comps/vxWorks` and are parsed by the project facility in alphabetical order. The files are written in component description language (CDL) which is described in the VxWorks BSP Developer’s Guide, 5.5 (for Tornado users) or the VxWorks Kernel Programmer’s Guide: Kernel (for VxWorks 6.x users).

3.2 Porting Drivers

This section provides some guidelines for porting drivers from earlier versions of VxWorks and from other operating systems.

3.2.1 Porting a Driver From An Earlier Version of VxWorks

In general, well-written drivers do not require much modification when being ported from older releases of VxWorks. Often the driver functions correctly after being simply “dropped” into the BSP. However, this depends on the VxWorks release involved, as well as the particular driver.

3.2.2 Porting a Legacy Driver to the VxBus Model

Porting a legacy VxWorks driver to be VxBus compliant involves several simple changes. An overview of the porting process is provided in the steps below. For more information, see 10.1 About VxBus, p.275.

Step 1: Register With VxBus

The first step in porting a driver to VxBus is to get the driver registered with VxBus. This involves creating the recognition structure for each bus type on which the device could be found and creating a set of entry points for the initialization. In early development, the entry points for the initialization routines can be empty. At this stage of development, you should also create a CDF file for the driver.
Step 2: Access Registers Through VxBus

Next, convert access to device registers into the appropriate VxBus calls or macros. Decide which portions of the driver should have code for optimized access to device registers. In these cases, replace the existing code with macros. Note that these macros may be expanded to inline code in the BSP, therefore use them only where necessary. Excessive use will result in unnecessarily large driver object modules.

Use an additional set of macros for the non-optimized register accesses. Create default macro values that contain the existing access code and replace the existing direct register accesses with calls to the macros. Once this work is complete, test the driver to verify that it appears in VxBus and that the driver works using the old access methods with the new macros.

Step 3: Modify BSP Initialization

Now, modify the BSP so that it does not initialize the device and the driver, and make sure that the driver’s VxBus registration routine does not make a call to `vxbDevRegister()`.  

Step 4: Convert Macros to VxBus Equivalents

Replace the macros for standard accesses with the VxBus equivalents, build the VxWorks image, and boot it. The driver should not show up as a registered driver because the registration routine does not call `vxbDevRegister()`. 

Step 5: Debug Registers

Attach a debugger, set a breakpoint at the initialization routine if required, and make the appropriate `vxbDevRegister()` call from the debugger or the shell. Step through the initialization code, verifying that the correct register is being used in each case.

NOTE: You may encounter difficulties if your driver refers to device registers through a structure instead of through the recommended base plus offset method. For these drivers, you should consider converting to the base plus offset method before porting to VxBus.

3.2.3 Porting From Another OS

Precise instructions for porting drivers from other operating systems are beyond the scope of this book. However, the tips provided in the remainder of this section apply in most cases.
3 Adding Drivers to VxWorks

3.3 Creating a New Driver Using the Legacy Model

Writing a driver typically consists of two separate parts: the interface to the device registers and functionality, and the interface to the OS. When porting a driver from another OS, the OS interface portion generally requires heavy modification or, in some cases, must be rewritten from scratch. This is not the case with the device interface.

Device-specific access routines are often the most difficult part of creating a device driver. These routines, and even snippets of code, can often be extracted from a working driver for another OS and put into a VxWorks driver. When doing this, you may find it useful to start from a template VxWorks driver of the same type. You can also check the Wind River Online Support Web site for an OS interface routines template for your particular driver type. If a template is not available, you can use the OS interface from an existing driver.

3.3 Creating a New Driver Using the Legacy Model

This section provides a basic overview of the steps required to create a new driver using the legacy (non-VxBus) model. (For information on creating VxBus compliant drivers, see 10. VxBus.)

The instructions in this section are not comprehensive and further information for specific driver types is available in later chapters of this document.

3.3.1 Step-By-Step Instructions

In general, Wind River recommends that you design from the top-down, but implement and test from the bottom-up. These techniques are described in the following sections.

Top-Down Design

Top-down design implies planning and documenting before starting any actual coding. To implement this design plan, add the elements described in the following sections in turn.
Step 1: Locate a Template File

If possible, start from a template file. If a template file is not available for the driver you are developing, start with an existing driver. However, you should be aware that starting from an existing driver often introduces problems from the other driver. Wind River provides template files for all classes of drivers.

Step 2: Add a Module Description

Next, find the chip manual for your device and copy the introductory description of the chip into the module description body. You should add paragraphs detailing how you intend your driver to work. For example, document which parts of the chip are controlled and what operating modes are supported.

Step 3: Define the Device Structures

Now, you must define a basic per-device data structure and a per-driver data structure. These structures define the state of the driver and each device. Recall that using structures allows the debugger to display the complete state of either the driver or the device with a single print command.

Step 4: Document the Macro Definitions

At this point, document any macros that can be used to customize the driver for different applications, such as chip-access macros, memory-access macros, and so forth. You should be sure to document what the macros do and how they are defined in the default situation.

Step 5: Declare Routines

Next, declare all of the routines that you think are required in order to use your driver. This includes routines the user or BSP may need to use the driver, as well as utility routines that are only used internally. Consult the documentation for your device and any available example drivers to ensure that basic functionality is covered by your chosen set of routines.

Note that it is usually best to design the driver such that you write a routine for each logical operation of the device. For example, if an SIO device clears the receive interrupt and loads the received character into a device register with a single command, it is best to write a single routine to clear the receive interrupt and read the character from the device register with the same call. Later development will be more difficult if you write one routine to clear all transmit, receive, and error interrupts, and a second routine to read the received character.
Step 6: Block Out Your Routines

Now, create the banners and declarations for all the routines you plan to write. You can leave the bodies empty until you are ready to write code. Write your routine description in the comment block. The comments in the comment block are more important than those in the code body. Comment blocks help others study the design without having to read the code.

Bottom-Up Implementation

After planning your driver and laying out a skeleton of routines and data structures, you are ready to implement and test.

Step 1: Write the Code

Start writing the code for your driver beginning with device initialization. Then, write code that accepts and initializes a device structure. Fill in the bodies of all other low-level driver routines. At this point, you can do a test compile to check that all of the necessary routines are declared. Examine the symbols in the object module to verify that only the desired external routines are unresolved.

Step 2: Test, Debug, and Recompile

Now, you can begin your test, fix, and recompile cycle. Personal preferences guide this phase but some developers choose to code and test one routine at a time.

Step 3: Work One Layer at a Time

Focus on the lower-level features (or layers) of your device driver before moving on to higher-level, and ancillary features. For ease of development, you should focus on one functional area at a time rather than attempt to complete the entire driver all at once. A sample work order is provided below. Note that the order of the steps is not critical and the descriptions of the layers may not be applicable to your device:

1. Get the operating system to recognize that your driver exists and be sure the driver can interact with the I/O system. Note that you do not need the actual device to be present on the system for this effort.
2. Define and implement routines that perform low-level device operations.
3. Implement and debug the top-level ISR code to handle each of the interrupt conditions that the device can generate. The initial version of these routines can be comprised of stubs that disable the interrupt source and print a message to the console using logMsg().
4. Implement code to perform I/O operations on the device, such as read and write operations, and otherwise manipulate the device as appropriate for the class of driver being developed.

5. Implement appropriate ioctl() calls to ensure that the driver fits into the I/O system well and perform general cleanup (that is, be sure your driver is at an acceptable release level).

**Step 4: Complete Performance Testing**

As one of the last phases of development, you should use a set of benchmark tests to verify that the device meets typical user expectations. For example, a wide SCSI device that can only deliver 5 MB/s net throughput is probably not acceptable for a user. You may need to write a complete performance benchmark test as part of the overall device driver development project.

**Helpful Hints**

The following information may be useful when designing and implementing your device driver.

**Avoid printf() in Drivers**

Avoid using the printf() routine in drivers, even for debugging purposes. Instead, use the logMsg() routine. There may be system interactions between the driver and the I/O system that can cause printf() to crash the system.

**Calling intConnect()**

Do not call the intConnect() routine—or any other routine that calls malloc()—from within sysHwInit(). The memory partition library is not initialized at this point in the system bring-up and the system will crash.

**3.3.2 Cache Considerations**

The VxWorks cache library (cacheLib) is designed to hide architecture and target details from the rest of the system and to provide mechanisms to maintain cache coherency. Cache coherency means data in the cache must be in sync (or coherent) with that in RAM.

Cache issues affect only those devices that can access memory shared with the CPU. If your device is not capable of performing read or write accesses directly to
Device drivers are one of the module types in a VxWorks system that can experience problems with data cache coherency. (Note that the CPU instruction data cache coherency is maintained elsewhere in the system.) This section describes cache issues you may encounter during driver development and how to use cacheLib to deal with these issues.

This section also describes how to enable the virtual memory library. This is necessary if you are using certain architectures. In these architectures, the VxWorks cache library controls the cache by calling the virtual memory library to manipulate the memory management unit (MMU).

NOTE: If your target architecture includes an MMU, and you disable virtual memory support but enable the data cache, the cacheDmaMalloc() routine cannot provide buffers that are safe from cache coherency issues.

NOTE: If you use an MMU, the cache modes are controlled by the cache mode values in the sysPhysMemDesc[] table and not by the configuration macros USER_I_CACHE_MODE and USER_D_CACHE_MODE. Assuming that these macros always control the cache operating mode is a common mistake.

Using the Cache Library

Before reading this section, review the reference entry for cacheLib. This section describes how to maintain data cache coherency for device drivers by using the VxWorks cache library (cacheLib). It also describes the cacheLib mechanism for controlling the side effects of CPU write piping, and provides additional hints for handling cache-related issues.

Review of cacheLib Facilities

The cacheLib reference entry describes the library facilities in detail. This section provides a brief definition of the facilities discussed in this document. Recall that the cacheLib.h header file contains the routine prototypes and macros required to make use of the cacheLib facilities. You should include the cacheLib.h file in your driver code.
This document also references the following routines and macros:

- The `cacheDmaMalloc()` routine allocates memory from the system and returns a pointer to the physical address. This routine attempts to use the underlying hardware features to provide a memory region that is safe from cache coherency issues. This memory region is guaranteed to start on a cache line boundary, and to not overlap cache lines with adjacent regions.

- The `CACHE_DMA_xxxx` macros (such as `CACHE_DMA_FLUSH`) flush, invalidate, and learn attributes of memory regions provided by the `cacheDmaMalloc()` routine.

- The `CACHE_DRV_xxxx` macros (such as `CACHE_DRV Invalidate`) flush, invalidate, and learn the attributes of ordinary memory (that is, memory on the device itself and device registers mapped into memory space, not necessarily cache-coherent) that the driver controls.

- The `CACHE_USER_xxxx` macros (such as `CACHE_USER_IS_WRITE_COHERENT`) flush and invalidate user memory that is outside the domain of the driver.

**Conducting Driver Analysis**

Each driver has a set of attributes that define its behavior relative to the cache. This section provides some guidelines for analyzing your driver for potential cache coherency problems. When you know the driver’s attributes, you can create a strategy for use of the `cacheLib` routines in the driver.

Before proceeding, determine if your device is capable of direct memory access (DMA). If the device is not capable of performing read or write accesses directly to memory that is shared with the CPU, your driver may not need any of the cache-related facilities of the `cacheLib`. Cache issues affect only those devices that can access memory shared with the CPU.

If the CPU architecture performs buffered writes, you may need to deal with `WRITE_PIPING` even if the device does not include DMA. Device registers that are memory mapped should not be cached. In most hardware, there is a hardware mechanism provided that keeps I/O addresses from being cached. However, you should keep in mind that even a non-DMA type device can still have issues related to write pipelining (see `Driver Attributes`, p.30).
Shared Memory Types

For DMA-type devices, the driver and the device share one or more memory regions. This shared memory can be one of the following types:

- Memory that is allocated using the `cacheDmaMalloc()` routine. This memory is associated with the `CACHE_DMA_xxxx` macros and is under the control of the driver and the underlying hardware with respect to cache issues.

- Memory that is allocated using the `malloc()` or `memalign()` routine or declared in the data or bss sections of the module (stack memory must never be shared with a device). This type of memory is associated with the `CACHE_DRV_xxxx` macros and is solely under the control of the driver with respect to cache issues.

Because you cannot control the positioning of data obtained by these methods, this type of memory has an inherent problem: the possibility of sharing a cache line with an adjacent region that does not belong to the driver. This means that flush and invalidate operations within this region can interfere with the coherency of the neighbor’s data in the shared cache lines. Therefore, restrict the use of this type of memory to exclude the first and last cache line in the region. Because the cache line sizes vary on different systems, this can be a portability issue.

By using `memalign()`, it is possible to ensure that buffers are cache line aligned at their starting address. If you need to protect the end of the buffer, you should increase the size of this request by at least one cache line size. This ensures that the end of the buffer does not share a cache line with any another buffer.

- Memory that is of unknown origin. This memory is associated with the `CACHE_USER_xxxx` macros and is outside the control of the driver. That is, the driver does not know how the memory is allocated, but the `CACHE_USER_xxxx` macros assume the memory is allocated using `malloc()`.

- Memory that is fixed or for a special purpose. This memory is a special region that is provided by the target hardware and is intended for the exclusive use of a DMA-type device. This type of memory is typically used for performance reasons. This memory is not part of the system pool, and thus is not associated with the `malloc()` routine, the `cacheDmaMalloc()` routine, the data section, or the bss section. There are no cache macros associated with this type of memory. The hardware usually provides a means to make this entire region non-cacheable.
Driver Attributes

This section lists the driver attributes you need to be aware of when planning your cacheLib strategy. Each attribute is given a name to simplify later discussion. Other than write pipelining, the attributes are of concern only to a DMA-type device.

WRITE_PIPING

The CPU uses write pipelining. Write pipelining means that CPU write operations are held in a pipeline until the external bus reaches some optimum state. Write pipelining is used on RISC architectures, such as MIPS. This technique can seriously impact your driver because it can delay the delivery of commands or data to the device your driver controls.

Set this attribute to TRUE if the driver is going to be run on a CPU that has a write pipe.

USER_DATA_UNKNOWN

The user data is in an unknown state. This attribute is set to TRUE if your driver passes data by address directly between the device and the driver’s user. In this case, data passed to the driver by the user is of an unknown cache state. An example of this is a pointer to a data buffer given to a SCSI disk driver for writing to the disk. In this case, the driver does not know if the buffer has been flushed to memory, or is still in cache. Conversely, data that is obtained from the device must be coherent between cache and memory before the pointer to that data can be given to the user for consumption.

MMU_TAGGING

The hardware provides MMU tagging of cache regions. This attribute is set to TRUE if an MMU that allows tagging of memory regions as non-cacheable is available on the target hardware.

DEVICE_WRITESASYNCHRONOUSLY

The hardware provides asynchronous write operations to shared memory. This attribute is set to TRUE if the device can perform write operations to shared memory asynchronously to driver activity.

SHARED_CACHE_LINES

The hardware allows both the driver and the device to write to a single cache line. If any single cache line can be written by both the driver and the device, this attribute is set to TRUE. An example of this is a shared data structure such as the following:
struct
{
    short command;
    short status
};

In this case, the driver is responsible for writing the command field and the device is responsible for writing the status field.

**SNOOPED**

The hardware provides bus snooping. This attribute is set to TRUE if the target hardware supports bus snooping. Bus snooping makes cache issues transparent to the driver. Only full bus snooping that maintains full coherency is supported. Some processors provide partial snooping that does not meet the requirements of full memory coherency. Without full snooping, this attribute must be FALSE.

**SHARED_POINTERS**

The device’s control model allows the driver and the device to exchange memory pointers. This attribute is set to TRUE if the driver and the device exchange memory pointers.

### Developing a cacheLib Strategy

This section describes how to devise a cacheLib strategy based on your driver's attributes.

Before proceeding, establish the range of hardware that the driver supports—for example, a single board, multiple differing boards, or general support. This decision affects your choice of cache strategy. For example, a driver intended for broad support can be restricted to certain cacheLib routines because board specifics are unknown. Alternatively, drivers intended to support only a single known board can be designed to use the cacheLib routines that make the best use of the target hardware.

**NOTE:** The level of cache library support designed into a driver must be well documented in the code.

Many of the routines and macros in cacheLib are designed to abstract the details of the underlying hardware. The macros can actually call routines within cacheLib, or they can do nothing at all. Their action depends on how certain function pointers are initialized at run-time. The run-time initialization of cacheLib is determined by the code in the BSP modules. The actual details of the
caching hardware are known in these BSP modules. This abstraction of details allows drivers to be broadly targeted, which increases portability.

One goal of driver design is portability. The following subsections assume portability is your goal and only minimally discuss target-specific applications.

**Flush and Invalidate Macros**

Generally, you need to add flush and invalidate macros to your driver. The macro set you implement depends on the type of memory you need to control (see *Shared Memory Types*, p.29). The flush macro is used on memory locations that are given to the device for subsequent reading by the device. The invalidate macro is used before the driver reads from memory locations that are written by the device.

Portable drivers use flush and invalidate macros as necessary. The macros are written to perform null operations if they are not needed on a particular target.

**WRITE_PIPING** Attribute

The `cacheLib` macro `CACHE_PIPE_FLUSH` flushes the write pipeline of the CPU. Typically, you add this macro to appropriate locations throughout your driver code. An appropriate location for this macro is after one or more commands have been written to the device. Device commands that require immediate delivery are most important. These type of commands are commonly found in initialization sequences or in interrupt handlers (where an interrupt condition requires clearing).

In order for you to decide when this macro needs to be used, you must have in-depth knowledge of the device you are programming. Some devices are not affected by the delays caused by a write pipeline. Such devices do not require the pipeline flush macro in the driver. Other devices do not function correctly without a tight synchronization with the driver. Such devices require liberal use of the pipeline flush macro throughout the driver.

Wind River recommends that you use `CACHE_PIPE_FLUSH` between an I/O write operation and a following I/O read operation. Successive writes or reads should not require any special protection. Pipeline flushing becomes essential when switching from writing to reading operations.

**NOTE:** Flushing the write pipeline does not automatically flush the cache to memory. You must still handle cache issues in your driver. Conversely, flushing or disabling the cache does not automatically flush or disable the write pipeline. The write pipeline is first in the hierarchy.
SNOOPED Attribute
This attribute is transparent to all memory types. Portable drivers use the appropriate flush and invalidate macros. However, if SNOOPED is set to TRUE, the macros evaluate to null operations. Target-specific drivers do not need flush or invalidate macros.

**NOTE:** Some systems implement partial snooping. They can read snoop, and only invalidate cache data on a write. This does not meet the requirements for fully coherent memory. In this case, the SNOOPED attribute is set to FALSE.

MMU_TAGGING Attribute
This attribute is transparent to the memory type obtained using the cacheDmaMalloc() routine. Portable drivers use the CACHE_DMA_FLUSH and CACHE_DMA_INVALIDATE macros. However, if MMU_TAGGING is TRUE, these macros evaluate to null operations. Target-specific drivers do not need CACHE_DMA_FLUSH or CACHE_DMA_INVALIDATE macros.

This attribute does not typically affect memory obtained using the malloc() routine, the data section, or the bss section. Therefore, appropriate use of the CACHE_USER_xxxx and CACHE_DRV_xxxx flush and invalidate macros is required, as described in Flush and Invalidate Macros, p.32.

USER_DATA_UNKNOWN Attribute
With this attribute, your driver must maintain cache coherency of any data that is passed directly between the device and the user. For outgoing data, first flush the user data from the cache with the CACHE_USER_FLUSH macro before commanding the device to access the data. For incoming data, you must first invalidate the cache for the device data using the CACHE_USER_INVALIDATE macro before passing the data to the user.

DEVICE_WRITESASYNCHRONOUSLY Attribute
You need only be concerned with this attribute when it is combined with the SHARED_CACHE_LINES attribute; see the SHARED_CACHE_LINES and DEVICE_WRITESASYNCHRONOUSLY/SHARED_CACHE_LINES attribute descriptions below.

SHARED_CACHE_LINES Attribute
The driver can easily function with this attribute as long as it can synchronize all operations to the data fields that may share a cache line (if the driver cannot synchronize these operations, see the following section). If the device cannot write to the data fields until explicitly commanded by the driver, you must sequence the use of flush macros, invalidate macros, and device commands in your driver in a manner that ensures data integrity.

NOTE: Some systems implement partial snooping. They can read snoop, and only invalidate cache data on a write. This does not meet the requirements for fully coherent memory. In this case, the SNOOPED attribute is set to FALSE.
DEVICE_WRITESASYNCHRONOUSLY and SHARED_CACHE_LINES Attributes

A combination of DEVICE_WRITESASYNCHRONOUSLY and SHARED_CACHE_LINES attributes can cause problems in your device driver. Because your driver cannot synchronize the write operations of the device, the potential exists for data corruption. Data corruption occurs when the device performs a write operation and the driver then performs a flush operation. In this sequence, the data value written by the device is wiped out by the cache line flush operation.

For this set of attributes, your driver cannot use the general CACHE_DRV_FLUSH and CACHE_DRV_INVALIDATE macros because the driver can no longer control the integrity of the shared memory. The shared memory must be obtained using the cacheDmaMalloc() routine. Furthermore, because the driver cannot perform any flush operations, this routine must provide memory that is write-coherent. Your driver can check the attributes of the memory provided by cacheDmaMalloc() using the CACHE_DMA_IS_WRITE_COHERENT macro. If this macro evaluates to FALSE, the driver cannot function reliably and should abort.

SHARED_POINTERS Attribute

This attribute is only a problem for the device driver if the memory shared between the driver and the device is obtained using the cacheDmaMalloc() routine. Because this routine can invoke MMU or other special hardware services, the address of the provided memory can be a virtual address. The device, however, can deal only in physical addresses.

In this case, use the CACHE_DMA_VIRT_TO_PHYS and CACHE_DMA_PHYS_TO_VIRT macros to convert between driver-viewed addresses and device-viewed addresses.

Additional Cache Library Hints

Additional hints for using the cache library are:

- Before adding any cacheLib support to your driver, ensure that the driver works reliably with the data cache turned off and that you do not experience cache coherency problems with the data cache disabled. When you begin to add cacheLib support, you can enable the data cache. Any changes to the driver's reliability can then be attributed to cache issues.

- Ensure that the driver works with cacheLib support on a single target first, using target-specific knowledge to guide the strategy. Once you have support for a single known target, you can then modify the driver's cacheLib strategy for broader, target-independent support.
3.3.3 Hotswap Support

VxWorks 5.5 and 6.x do not support hotswap.

3.3.4 Downloadable Driver Support

In VxBus, all drivers can be downloaded to the system at any time. The driver is downloaded by some means and the driver’s registration routine is called. For example, during development, the driver can be downloaded by the debugger and the developer can call the driver’s registration routine through the debugger or through the shell. The driver then participates in the three-stage VxBus device initialization process (see 10.6 Multi-Stage Initialization, p.283).

This does not account for the interface between the driver and operating system or middleware modules that might make use of the driver. For most operating system and middleware modules that are VxBus compliant, the additional configuration can be initiated manually by calling the \texttt{vxbDevMethodGet()} routine and then calling the appropriate initialization routine using the function pointer that is returned by \texttt{vxbDevMethodGet()}. Or, the function pointer can be called directly by the driver.

Because the full VxBus initialization sequence is followed regardless of when the driver is loaded, it is possible to design an application-specific driver management mechanism that keeps copies of all device driver object modules on a backing store. This driver manager would load the driver into RAM only when a device that requires the driver is discovered.
4.1 Introduction

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

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

If this is the first time you have written a network interface driver, consider taking an existing driver and modifying it to meet your needs. For more information on TCP/IP, Wind River recommends the following Addison Wesley publications:
4.2 END Driver Overview

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

4.2.1 Driver Environment

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

The MUX

The multiplexor (usually known as the MUX, and referred to as the MUX in this document) is an interface that joins the data link and protocol layers. An END driver does not directly interface with the data link layer, but rather interfaces with the MUX, which is an abstraction layer that is intended to de-couple the END driver from any particular protocol. This API multiplexes access to the networking hardware for multiple network protocols. Figure 4-1 shows the MUX in relationship to the protocol and data link layers.

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

**Figure 4-1** The MUX Interface Between Data Link and Protocol Layers

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

**Network Interface Drivers and Protocols**

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

For example, after receiving a packet, the network interface driver does not directly access any structure within the protocol. Instead, when the driver is ready to pass data up to the protocol, the driver calls a MUX-supplied routine. This routine then handles the details of passing the data up to the protocol.
The purpose of the MUX is to de-couple the network driver from the network protocols, thus making the network driver and network protocols nearly independent from each other. This independence makes it much easier to add new drivers or protocols. For example, if you add a new END driver, all existing MUX-based protocols can use the new driver. Likewise, if you add a new MUX-based protocol, any existing END driver can use the MUX to access the new protocol.

The MUX, Protocol, and Driver API

Figure 4-1 shows a protocol, the MUX, and a network interface driver. The protocol implements the following entry points:

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

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

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

The network interface driver implements the entry points endLoad(), endUnload(), endSend(), and so forth. The MUX uses these entry points to interact with the network interface driver. When writing or porting a network interface driver to use the MUX, you must implement all of the entry points listed in Table 4-4 in Required Driver Entry Points, p.102.
In Figure 4-2, the arrows indicate calls to an entry point. For example, the top-most arrow tells you that the protocol calls `muxBind()`, a routine implemented in the MUX. If the MUX-based API specifies both ends of the call, the figure specifies a routine name at each end of an arrow. For example, `muxSend()` calls `endSend()`. Note that although the protocol started the send by calling `muxSend()`, the figure does not name the protocol routine that called `muxSend()`. That routine is outside the standardized API.
Driver Components

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

An END driver's basic components include:

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

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

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

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

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

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

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

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

The larger the END driver operating bandwidth, the greater its memory requirements. Occasionally, an END driver does not have sufficient memory resources to accommodate the data inflow. This can be due to system constraints, buffer loaning, or CPU starvation. When a driver gets into an insufficient resource condition, it continues to provide the DMA engine buffers into which inflowing data is transferred but the driver does not pass these buffers up to the stack.
A protocol requests that an END driver transmit a frame by calling the `muxSend()` routine, which in turn calls the driver’s registered send routine. Sends can occur at any time, and may occur before previous sends have completed.

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

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

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

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

### Protocols That Use the MUX API

This section describes how to port protocols to the MUX-based model. As shown in Figure 4-1, MUX-based protocols bind themselves to the MUX from above and network interface drivers (END drivers) bind themselves to the MUX from below. Thus, a protocol is layered on top of the MUX, which is layered on top of a network interface driver. The responsibilities of each are summarized below.
Protocol:
- Interface to the transport layer, and through it, to the application programs.
- Usually, acts as a source of transmit packets and a sink of received packets.
- Returns buffer resources from received packets to the driver pools.

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

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

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

Protocol Startup

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

As input to `muxBind()`, you must specify the base name and unit number of a network device (for example, `ln` and `0`, `ln` and `1`, `ei` and `0`, and so on), as well as the appropriate receive, transmit restart, and shutdown routines for the protocol; a protocol type, and a name for the attaching protocol.
There are three special protocol type values, as well as the normal network-layer protocol type values from RFC 1700, corresponding to the Ethernet header type field. The three special type values are MUX_PROTO_OUTPUT, MUX_PROTO_SNARF, and MUX_PROTO_PROMISC. MUX_PROTO_OUTPUT is used for output protocols—which are passed packets in the send path, but not the receive path. There may be no more than one output protocol for a given interface. (Output protocols are discussed further below). MUX_PROTO_SNARF protocols, normal “typed” protocols, and MUX_PROTO_PROMISC protocols attached to an END interface may be delivered packets received on that interface.

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

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

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

### Output Protocols

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

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

Receiving Data

In response to an interrupt from the network device, VxWorks executes the device’s previously registered interrupt service routine. This routine gets the packet off the device and queues it for processing the task level, where the driver prepares the packet for hand-off to the MUX. For a more detailed description of this process, see Handling Packet Reception, p. 69.

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

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

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

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

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

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

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

Protocol Shutdown

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

Interactions With the MUX API

This section presents the routines and data structures that the protocol uses to interact with the MUX. Most of the work is handled by the MUX routines (listed in Table 4-1). Unlike the driver entry points described earlier, you do not implement the MUX routines. These routines are utilities that you can call from within your protocol. For specific information on these MUX routines, see the appropriate API reference entry.

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

<table>
<thead>
<tr>
<th>Table 4-1 MUX Interface Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUX Routine</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td><code>muxDevLoad()</code></td>
</tr>
<tr>
<td><code>muxDevStart()</code></td>
</tr>
<tr>
<td><code>muxBind()</code></td>
</tr>
</tbody>
</table>
Table 4-1  MUX Interface Routines

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

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

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

Passing a Packet Up to the Protocol: stackRcvRtn()

Each protocol must provide the MUX with a routine that the MUX can use to pass packets up to the protocol. This routine must take the following form:
void stackRcvRtn
(
    void* pCookie, /* returned by muxBind() call */
    long type, /* protocol type from RFC 1700 */
    M_BLK_ID pNetBuff, /* packet with link level info */
    LL_HDR_INFO* pLinkHdr, /* link-level header info structure */
    void* pSpare /* a void* the protocol can use to get info */
    /* on receive. This was passed to muxBind(). */
)

Your protocol must declare its \texttt{stackRcvRtn}() as \texttt{void}. Thus, this routine returns no value.

The parameters are:

\textbf{pCookie}

Expects the pointer returned from the \texttt{muxBind()} call. This pointer identifies the device to which the MUX has bound this protocol.

\textbf{type}

Expects the protocol type from RFC 1700 or the SAP.

\textbf{pNetBuff}

Expects a pointer to an \texttt{mBlk} structure that contains the packet data and the link-level information.

\textbf{pLinkHdr}

Returns an \texttt{LL_HDR_INFO} structure containing header information that is dependent upon the particular data-link layer that the END driver implements. For more information, see \textit{Tracking Link-Level Information: LL_HDR_INFO}, p.100.

\textbf{pSpare}

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

\textbf{Passing Error Messages Up to the Protocol: stackError()} \texttt{()}

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

\begin{verbatim}
void stackError
(
    END_OBJ* pEnd, /* pointer to END_OBJ */
    END_ERR* pError, /* pointer to END_ERR */
    void* pSpare /* pointer to protocol private data passed in muxBind */
)
\end{verbatim}
You must declare your `stackShutdownRtn()` as returning `void`. Thus, there is no returned function value for this routine. The parameters are:

**pEnd**

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

**pError**

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

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

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

- **END_ERR_INFO**
  
  This error is information only.

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

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

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

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

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

**pSpare**

Expects a pointer to protocol-specific data. Originally, the protocol passed this data to the MUX when it called `muxBind()`. This data is optional and protocol-specific.
Shutting Down a Protocol: \texttt{stackShutdownRtn()}

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

\begin{verbatim}
void stackShutdownRtn

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

}
\end{verbatim}

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

The parameters are:

\begin{itemize}
    \item \texttt{pCookie} \\
        Expects the pointer returned as the function value of the \texttt{muxBind()} for this protocol. This pointer identifies the device to which the MUX has bound this protocol.
    \item \texttt{pSpare} \\
        Expects the pointer passed into \texttt{muxBind()} as \texttt{pSpare}.
\end{itemize}

Restarting Protocols: \texttt{stackTxRestartRtn()} 

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

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

Your \texttt{stackTxRestartRtn()} must take the following form:

\begin{verbatim}
void muxTxRestart

    void* pCookie /* Returned by \texttt{muxBind()} call. */

}
\end{verbatim}
The parameters are:

pCookie

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

Network Layer to Data Link Layer Address Resolution

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

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

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

STATUS muxAddrResFuncDel
(
    long ifType, /* Media interface type from m2Lib.h */
    long protocol /* Protocol type from RFC 1700 */
)
```

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

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

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

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

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

Understanding How VxWorks Launces and Uses Your Driver

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

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

Launching Your Driver

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

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

Loading Your Driver into the MUX

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

The endLoad( ) routine handles any device-specific initialization and returns an END_OBJ structure. Your endLoad() routine must populate most of this structure (see Providing Network Device Abstraction: END_OBJ, p. 96). This includes providing a pointer to a NET_FUNCS structure populated with function pointers to your driver’s entry points for handling sends, receives, and so forth.

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

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

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

**NOTE:** A tuple is a construct used by the VxWorks stack and drivers to access and manage data buffers. A detailed description of a tuple is provided in Receive and Transmit Descriptors, p.61.

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

**Registering Your Driver’s Interrupt Routine**

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

**Using tNetTask**

When working with END drivers, it is necessary to understand the use of `tNetTask`, how it operates, and why to use it.
Many desktop and mainframe operating systems use network drivers that dispatch incoming packets directly to the application that receives the packets. This operation is done in the lower half of the OS, from within interrupt context. Therefore, much of the network stack is executed from within interrupt service routines (ISRs).

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

\textbf{Interrupt Handlers}

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

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

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

\begin{verbatim}
STATUS netJobAdd
(
    FUNCPTR routine, /* routine to add to netTask work queue */
    int param1,      /* first arg to added routine */
    int param2,      /* second arg to added routine */
    int param3,      /* third arg to added routine */
    int param4,      /* fourth arg to added routine */
    int param5       /* fifth arg to added routine */
)
\end{verbatim}

In your call to \texttt{netJobAdd()}, you should specify your driver’s entry point for processing packets at the task level. The \texttt{netJobAdd()} routine then puts the function call (and arguments) on the \textit{tNetTask} work queue. VxWorks uses \textit{tNetTask} to handle task-level network processing.
There are several limitations on network interrupts in VxWorks. These limitations impact the way drivers are written.

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

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

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

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

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

`netJobRing` has a limited amount of space. Because of this, it is critical that the driver make efforts to conserve space on the `netJobRing`. If the ring is allowed to overflow, the network stack can become corrupt and the system may require a reboot. To safeguard against overflow, the driver must limit the number of jobs that it simultaneously places on the ring. This limit can be imposed through the use of queuing indicators. These indicators communicate to the driver if a particular

---

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

**CAUTION:** Use `netJobAdd()` sparingly. The `netJobRing` is a finite resource that is also used by the network stack. If it overflows, this implies that the network stack is corrupted.
interrupt handler is already queued on the ring. If the handler is already queued, it is not practical to queue it again before it has run. The indicators are fields in the driver control structure. There should be one indicator for the receive handler and another for the packet-complete interrupt. These indicators are discussed further in Receive Handler Interlocking Flag, p.74 and Transmit-Packet-Complete Handler Interlocking Flag, p.82, respectively.

- **Interrupt Masking**

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

### Executing Calls Waiting In the Network Job Queue

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

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

It is possible to design a driver that starves the network stack and other drivers. When a driver uses `taskDelay()` , or any other delay mechanism, in code executed in the context of `tNetTask`, the delay prevents processing of packets from other interfaces. For this reason, you must carefully consider the use of delays in the driver. Consider rescheduling the job with another `netJobAdd()` call instead of delaying with `taskDelay()` . This allows other interfaces, as well as the network stack, to perform other work while the driver is waiting.
Because interrupts are relatively costly in terms of overall system performance, one recommended goal of network device drivers is to process as many packets as possible before exiting. However, to avoid starvation of other interfaces, there should be a cap on the number of packets processed at any one time. If additional packets are available when the cap is reached, the driver can re-schedule the receive routine with another call to `netJobAdd()`.

### Adding Your Network Interface Driver to VxWorks

Adding your driver to the target VxWorks system is much like adding any other application. The first step is to compile and include the driver code in the VxWorks image. For a description of the general procedures, see the *Tornado User’s Guide* (for Tornado users) or the *Wind River Workbench User’s Guide* (for VxWorks 6.x users), as well as the *VxWorks Programmer’s Guide* (VxWorks 5.5) or the *VxWorks Kernel Programmer’s Guide* (VxWorks 6.x). These documents provide information on how to compile source code to produce target-suitable object code.

In addition to including the object module in the VxWorks image, you must do some additional work to initialize the END driver and get the MUX to recognize it. All Wind River VxWorks 6.x BSPs support an END driver. However, if the BSP you are using does not already include END driver support, you need to create a table of configuration information for END drivers, called `endDevTbl[]`. Once this is accomplished, you must populate the table with information about your driver and make sure your BSP calls the appropriate initialization routines. This is usually done in the file `configNet.h` in the BSP directory.

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

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

If the BSP already supports an END driver, the BSP should already contain the `endDevTbl[]` and appropriate macros. In this case, the `endDevTbl[]` table must be modified to include the new driver and you must create definitions containing the configuration information (this is typically done with `#define` statements, grouped together in one location in `configNet.h`).
In addition, VxWorks drivers are typically written to be independent of the bus and processor configuration. This means that the methods used to access device registers are provided by the BSP and not by the driver. For each supported driver, there is typically a `sysDev` file containing the definitions and routines necessary for the driver to get access to the device registers, interrupt connection code, and other resources. When adding a new driver to a BSP, this file must be provided.

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

```c
/* Parameters for loading the driver supporting buffer loaning. */
#define LOAD_FUNC_0 ln7990EndLoad
#define LOAD_STRING_0 "0xfffffe0:0xffffffe2:0:1:1"
#define BSP_0 NULL

/* Parameters for loading the driver NOT supporting buffer loaning. */
#define LOAD_FUNC_1 LOAD_FUNC_0
#define LOAD_STRING_1 "0xffffee0:0xfffffee2:4:1:1"
#define BSP_1 NULL
```

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

**END_LOAD_FUNC**

Specifies the name of your driver’s `endLoad()` entry point. For example, if your driver’s `endLoad()` entry point is `fei82557EndLoad()`, you must edit `config.h` to include the line:

```c
#define END_LOAD_FUNC fei82557EndLoad
```

**END_LOAD_STRING**

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

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

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

```c
END_TBL_ENTRY endTbl
{
  ( 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, FALSE),
  ( 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, FALSE),
  ( 0, END_TBL_END, 0, NULL),
};
```
The number at the beginning of each line specifies the unit number for the device. The first line specifies a unit number of 0. Thus, the device it loads is `deviceName0`. The FALSE at the end of each entry indicates that the entry has not been processed. After the system has successfully loaded a driver, it changes this value to TRUE in the run-time version of this table. If you want to prevent the system from automatically loading your driver, set this value to TRUE.

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

### Allocating, Initializing, and Utilizing Memory Resources

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

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

### Driver Control Structure

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

### Receive and Transmit Descriptors

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

---

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

The device uses the descriptors to:

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

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

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

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

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

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

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

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

Receive descriptors are typically organized as a ring. Both the device and the driver follow this ring and use or service the ring’s descriptors, respectively. The driver follows the device’s access, servicing the descriptors the device uses. When the device uses a descriptor, it sets the descriptor’s fields to indicate that its associated buffer has received DMA data. When the driver services the descriptor, it removes the filled buffer and replaces it with an empty one. The driver obtains the replacement buffer from its pool. It then clears the descriptor to indicate to the device that the descriptor is again ready for use. When the ring is first initialized, all descriptors have empty buffers and are ready for use.
NOTE: A complication to buffer replacement is that some architectures only read data on a four-byte boundary. An Ethernet header is 14 bytes long. If a DMA buffer is four-byte aligned, then the IP header is two-byte aligned. This results in an alignment mismatch. To compensate for this issue, the driver can offset the DMA buffer pointer in the descriptor by two bytes in order to put the IP header, and subsequent data, at four-byte boundaries. There is a further complication in that this solution requires the device to restrict DMA to a two byte address alignment. Not all devices support DMA using a two byte alignment. Therefore, a device that cannot perform a DMA write to a two-byte boundary cannot work with an architecture that cannot read from a two-byte boundary without copying the data to a new buffer to adjust the packet alignment.

Association List

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

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

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

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

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

Setting Up and Using Memory for Receive and Transmit Buffers

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

mBlks, cBlks, and Cluster Buffers

Included with the network stack is netBufLib, a library that you can use to set up and manage a memory pool specialized to the buffering needs of networking.
applications such as END drivers and network protocols. To support buffer loaning and other features, netBufLib routines deal with data in terms of mBlks, clBlks, and clusters.

The netBufLib routines use the mBlk and clBlk structures to track information necessary to manage the data in the clusters. The clusters contain the data described by the mBlk and clBlk structures. These elements—mBlks, clBlks, and cluster buffers—constitute a tuple. The mBlk structure is the primary vehicle through which you access or pass the data that resides in a tuple. Because an mBlk merely references the data, this lets network layers communicate data without actually having to copy the data. Another mBlk feature is chaining. This lets you pass an arbitrarily large amount of data by passing the mBlk at the head of an mBlk chain. See Figure 4-3.

Figure 4-3 Presentation of Two Packets to the TCP Layer

![Diagram of two packets to the TCP layer]

The netBufLib library provides two means of creating a network memory pool—the routines netPoolInit() and netPoolCreate(). The routines differ in that
netPoolInit() requires the user to allocate the memory used for the tuples. netPoolCreate() takes as arguments, attributes describing the characteristics of the pool’s memory and allocates and manages the memory on behalf of the user. This is a great advantage because it provides the driver with properly aligned and cacheable cluster buffers. Wind River highly recommends that you use netPoolCreate() instead of netPoolInit().

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

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

Setting Up a Memory Pool

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

All memory pools are organized around pools of tuples. However, because a network protocol typically requires clusters of several different sizes, its memory pool must contain several tuple pools (one tuple pool for each cluster size). In addition, each cluster size must be a power of two. Common cluster sizes for this style of memory pool are 64, 128, 256, 512, 1024, and 2048 bytes. See Figure 4-4.
By contrast, a memory pool intended for an END driver typically uses only one cluster size and the cluster size is not limited to a power of two. Thus, you are free to choose whatever cluster size is most convenient, which is typically something close to the maximum transmission unit (MTU) of the network. A network’s MTU is typically 1500 bytes.

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

Establishing a Network Driver Pool

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

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

   ```c
   if (pDrvCtrl->pNetBufCfg = (NETBUF_CFG *) memalign (sizeof(long),
                     (sizeof(NETBUF_CFG) +
                     END_NAME_MAX)) == NULL)
     return (ERROR);
   
   bzero(pDrvCtrl->pNetBufCfg,sizeof(NETBUF_CFG));
   ```
2. Initialize the pName field.

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

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

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

   pDrvCtrl->pNetBufCfg->attributes = ATTR_AC_SH_ISR;

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

   pDrvCtrl->pNetBufCfg->pDomain = NULL;

5. Set the ratio of mBlks to clusters.

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

6. Use a NULL value to set the memory partition of mBlks to kernel.

   pDrvCtrl->pNetBufCfg->ctrlPartId = NULL;

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

   pDrvCtrl->pNetBufCfg->bMemExtraSize = 0;

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

   pDrvCtrl->pNetBufCfg->bMemPartId = NULL;

9. Allocate memory for the network cluster descriptor.

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

10. Initialize the cluster descriptor.

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

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

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


    free (pDrvCtrl->pNetBufCfg->pClDescTbl);
    free (pDrvCtrl->pNetBufCfg);
Handling Packet Reception

The list of END driver entry points (see Table 4-4) makes no mention of an `endReceive()` entry point. That is because an END driver does not require one. Of course, your driver must include code that handles packet reception, but the MUX never calls this code directly. Thus, the specifics of the code for packet reception are left to you.

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

Internally, your task-level packet-reception entry point should do whatever is necessary to prepare the packet for handing off to the MUX, such as ensuring data coherence. Likewise, this entry point might use a level of indirection in order to check for and avoid race conditions before it attempts to do any processing on the received data. When all is ready, your driver should pass the packet up to the MUX. To do this, it calls the routine referenced in the `receiveRtn` member of the `END_OBJ` structure (see Providing Network Device Abstraction: END_OBJ, p.96).

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

Receive Handler

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

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

The device cycles through the descriptor ring. When the device receives an incoming Ethernet frame, it receives it into its FIFO. The device then writes the frame into the buffer which it locates through the currently accessed descriptor. The prevalent method used for a device to write data into both the descriptors and the buffers, is direct memory access (DMA).
As the network device indexes around the descriptor ring, it tests each entry for availability. When the device receives a frame and finds an available descriptor, its DMA engine fills the associated buffer and sets a status flag in the descriptor indicating that the buffer is full.

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

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

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

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

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

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

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

Efficient Receive Loop

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

while((rbdStatus)
Obtaining a New Tuple

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

```c
if ((pNewMblk = netTupleGet(pDrvCtrl->endObj.pNetPool, CLUSTER_SIZE, M_DONTWAIT, MT_DATA, 0)) == NULL)
{
    endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_IN_ERROR);
    endM2Packet (&pDrvCtrl->endObj, NULL, M2_PACKET_IN_DISCARD);
    return (ERROR);
}
```

Retrieving and Replacing DMA Buffers

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

1. Retrieve the used tuple as follows:
   ```c
   pMblk = pDrvCtrl->pMblkList[pDrvCtrl->index];
   ```

2. Place a new tuple on the association list:
   ```c
   pDrvCtrl->pMblkList[pDrvCtrl->index] = pNewMblk;
   ```
   If the device supports DMA to a 2 byte offset, move the `mBlk` data pointer by 2 bytes:
   ```c
   pNewMblk->mBlkHdr.mData = pNewMblk->pClBlk->clNode.pClBuf + pDrvCtrl->offset;
   ```

3. Ensure cache coherency of the DMA buffers as follows:
   ```c
   DRV_CACHE_INVALIDATE (pNewMblk->pClBlk->clNode.pClBuf, CLUSTER_SIZE);
   ```

4. Convert the buffer virtual address to a physical address:
   ```c
   pBuffer = VIRT_TO_PHYS ((UINT32) pNewMblk->mBlkHdr.mData;
   ```

5. Update the receive descriptor:
   ```c
   xxxDescBufWrite ((&pDrvCtrl->pRxDescBase[pDrvCtrl->index], pBuffer, BUFFER_OFFSET);
   ```

6. Copy DMA length to `mBlk`:
   ```c
   pMblk->mBlkHdr.mLen = (xxxDescRead (&pDrvCtrl->pRxDescBase[pDrvCtrl->index]) & ~0xc000);
   ```
Clearing the Descriptor Status

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

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

Incrementing the index

Next, increment the index:

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

Sending a Received Frame to the Stack:

To pass a buffer up to the MUX, a driver calls `muxReceive()`, which in turn calls the protocol’s `stackRcvRtn()` routine (see Passing a Packet Up to the Protocol: `stackRcvRtn()`, p. 49). When control returns from `muxReceive()`, the driver can consider the data delivered and can forget about the buffers it handed up to the MUX. When the upper layers are done with the data, they free the buffers back to the driver’s memory pool. The macro, `END_RCV_RTN_CALL`, which is provided by Wind River, calls `muxReceive()`

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

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

Fair Access Bounding

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

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

```c
int loopcounter = pDrvCtrl->maxRxFrames; /* local variable */
/* in receive handler */
```
while ((rbdStatus != RBD_STATUS_FREE) && (--loopcounter > 0))
{
    /* Receive Loop */
}

if (rbdStatus != RBD_STATUS_FREE)
{
    /* Put this job back on the netJobRing and leave */
    if ((netJobAdd ((FUNCPTR) xxxRecvHandler, (int) pDrvCtrl, 0,0,0,0)) == ERROR)
    {
        /* Very bad!!! The stack is now probably corrupt. */
        logMsg("The netJobRing is full. 2\n",0,0,0,0,0);
        return;
    }
}
else
{
    pDrvCtrl->rxJobQued = FALSE;
}

Receive Handler Interlocking Flag

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

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

The work most often done by an interrupt’s task-level service routine involves servicing a queue. It is efficient to continue to service this queue for as long as work is available. Because service routines continue to execute as long as there is work
to do, scheduling another instance of a service routine while one is already running is unnecessary and redundant. Because of this, you should try to coalesce interrupts. This can be accomplished in software by masking an incident interrupt in its ISR and leaving that interrupt masked while the service routine is running. Then, before exiting the service routine, re-enable the interrupt.

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

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

Implementing Receive Handler Interlocking Flag

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

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

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

```c
/* Test if fei82557RecvHandler() is on netJobRing. */
if(!pDrvCtrl->rxJobQued)
{
    /* fei82557RecvHandler() is not on netJobRing so put it on. */
    if ((netJobAdd ((FUNCPTR) fei82557RecvHandler, (int) pDrvCtrl,
        0, 0, 0, 0)) != ERROR)
    {
        pDrvCtrl->rxJobQued = TRUE;
    }
    else
    {
        logMsg("The netJobRing is full. 1\n",0,0,0,0,0);
        I82557_INT_ENABLE(SCB_C_M);
        return;
    }
}
```

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

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

Receiver Stall Handling

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

- Some devices simply require that the next descriptor in the sequence be cleared. That is, the descriptor’s status must be set to free or available. In this case, the device automatically detects that the stall is cleared and resumes operation without any action on the part of the driver.
• Other devices place their receiver into a halted state by setting a bit in a control register. For this type of device, it is often required that, in addition to freeing the next descriptor, the driver must clear the control register bit before operation resumes.

**Interrupt Re-Enabling**

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

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

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

**Two-Tiered Polling**

The technique previously used for scheduling a receive handler in END drivers involved a single tier polling method, referred to as *interrupt stimulated polling*. Using the interrupt stimulated polling method, the device would receive an incoming packet into its DMA ring and generate an interrupt. The interrupt handler would then disable the device’s interrupt and schedule a receive handler to run `tNetTask`. This receive handler then polled all descriptors on the DMA ring for the original packet that caused the interrupt and any additional DMAs that occurred since that initial DMA. When the receive handler finished servicing all of the completed DMAs, it would re-enable the device’s interrupt and exit.
The intention of this method was to service the maximum number of packets possible for each interrupt. This method attempted to relieve the system of the overhead implicit with frequent interrupts. However, interrupt stimulated polling resulted in an interrupt occurring for almost all received packets. This imposed considerable overhead on the system when servicing the large number of interrupts associated with high traffic loads.

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

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

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

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

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

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

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

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

- **pollLoops**
  - The maximum times the outer loop can increment before terminating.
Operation Details

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

1. The Receive Handler obtains and tests the next descriptor to be serviced
   
a. If the next descriptor indicates it needs service (full), the receive handler enters the receive loop and the counter, pollCnt, is cleared
   
b. If the next descriptor is empty, the receive handler exits without changing the status, counters, the index, or pointers.

2. When the receive handler enters the receive loop, the receive loop continues to service its descriptors, until it encounters one of two conditions:
   
   - It encounters an empty descriptor.
   - It reaches the maximum packet boundary set by the fair access limit.

3. After exiting the receive loop, the receive handler tests if either of two conditions exist:
   
   - The counter, pollCnt, is less than the value of pollLoops.
   - The next descriptor indicates it needs to be serviced (full).

   If either condition is true, the receive handler’s behavior depends on the status of the next descriptor to be serviced.

   a. If the next descriptor to be serviced is empty, the while loop must have terminated because it encountered an empty descriptor. The following actions are taken:
      
      i. The receive handler increments pollCnt.
      
      ii. The receive handler places itself back on the netJobRing to be executed again.
      
      iii. The receive handler sets the pollDone flag to FALSE indicating a continuation of the outer loop of polling.

   b. If the status of the next descriptor is full, the receive loop must have terminated because it reached the maximum number of descriptors to be serviced before relinquishing operation. The following actions are taken:
      
      i. The receive handler clears pollCnt.
      
      ii. The receive handler then places itself back on the netJobRing to be executed again.
iii. The receive handler sets the pollDone flag to FALSE indicating that it will continue the outer tier of polling.

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

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

ii. The receive handler clears pollCnt.

iii. The receive handler sets pollDone to TRUE.

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

5. The receive handler exits.

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

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

   \[\text{BOOL} \quad \text{pollDone}; \quad /* \text{Flag indicating outer loop exit */}\]
   \[\text{UINT32} \quad \text{pollCnt}; \quad /* \text{polling counter */}\]
   \[\text{UINT32} \quad \text{pollLoops}; \quad /* \text{polling limit */}\]

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

   \[\text{UINT} \quad \text{maxRxFrames}; \quad /* \text{max frames to Receive in one job */}\]

3. In the driver’s endLoad() routine, specify the addition of the maxRxFrames parameter to the END_LOAD_STRING.

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

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

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

   \[\text{pDrvCtrl->maxRxFrames} = \text{pDrvCtrl->nRFDs} * 2;\]
   \[\text{tok = strtok} \_\text{r} \left(\text{NULL, "\*"}, \&\text{holder}\right);\]
   \[\text{if } ((\text{tok} != \text{NULL}) && (\text{tok} != (\text{char} *)-1))\]
   \[\text{pDrvCtrl->maxRxFrames} = \text{strtol} \left(\text{tok, NULL, 16}\right);\]

5. In the driver’s start routine, initialize the two-tiered polling fields in the DRV_CTRL structure.
pDrvCtrl->pollCnt = 0;
pDrvCtrl->pollLoops = 1;
pDrvCtrl->pollDone = FALSE;

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

   int loopCounter = pDrvCtrl->maxRxFrames;

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

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

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

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

   /* Put this job back on the netJobRing and leave */
   if ((netJobAdd ((FUNCPRTR) fei82557RecvHandler, (int) pDrvCtrl,
       0,0,0,0)) == ERROR)
       {
       /* Very bad!! The stack is now probably corrupt. */
       logMsg("The netJobRing is full. 2\n",0,0,0,0,0);
       I82557_INT_ENABLE(SCB_C_M);
       return;
       }
   else
       {
       pDrvCtrl->pollCnt = 0;
       pDrvCtrl->pollDone = TRUE;
       pDrvCtrl->rxJobQued = FALSE;
   }

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

   if (pDrvCtrl->pollDone)
       { 
       I82557_INT_ENABLE(SCB_C_M);
       }
Handling Packet Transmission

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

Transmit-Packet-Complete Handler Interlocking Flag

Transmit-packet-complete interrupts are typically used to allow the driver to return resources to the pool after a packet is transmitted. The frequency of these interrupts can be very high. Because of the high frequency at which these interrupts are generated, transmit-packet-complete interrupts can degrade system performance and overflow `netJobRing`. `Transmit Descriptor Clean-up`, p.84 includes a discussion of how to reduce the frequency of this interrupt. This section deals with how to prevent the transmit-packet-complete interrupt from causing a `netJobRing` overflow. The method used is essentially the same as that used for the receive handler interlocking flag (see `Receive Handler Interlocking Flag`, p.74).

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

Supporting Scatter-Gather

Scatter-gather is a DMA technique that allows for a single large block of data to be distributed—or scattered—among multiple buffers. The data can then be gathered together later and transferred in a single DMA transaction, as if it were stored in a contiguous buffer. This capability is desirable because the network stack is often unable to find a single cluster buffer that is large enough to hold a large packet. That is, when the network is unable to find a buffer of sufficient size, it must obtain multiple tuples with cluster buffers that, cumulatively, have sufficient space to hold the packet. The stack then fragments the packet among multiple tuples. For transmit, the fragmented packet is sent as an `mBlk` chain to the driver’s send routine to be transmitted.
When scatter-gather is not supported by the device and the driver is sent a fragmented packet, the driver must obtain a single buffer from its pool and must then copy the packet fragments into a single buffer. This is possible because the driver pool, unlike the network stack pool, typically has only a single buffer size that is sufficient to hold the largest packet the maximum transfer unit (MTU) allows. This means that in most cases, the driver can find a buffer that is large enough to accommodate any packet. However, the overhead of requiring the driver to obtain a buffer and copy the packet fragments into the buffer is a substantial drag on overall system performance.

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

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

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

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

**NOTE:** In END drivers, scatter-gather is not a concern for packet reception. This is because the driver’s buffers are all of a single size and are sufficient to hold the maximum incoming frame (MTU). Therefore, END drivers do not fragment incoming frames.
If the number of available descriptors is insufficient to hold the packet data, the send routine attempts to free enough descriptors to handle the packet. If the send routine fails to free a sufficient number of descriptors, it must then either coalesce the packet into a single buffer—the same practice that is used if scatter-gather is not supported—or it must throw the packet away.

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

**Transmit Descriptor Clean-up**

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

After a packet is successfully transmitted, most devices generate a packet-complete interrupt. The ISR for this interrupt causes the driver’s transmit-packet-complete handler to be scheduled, which in turn calls the driver’s transmit descriptor clean routine to free the packet descriptor or descriptors and the associated mBlk chain. As described in *Transmit-Packet-Complete Handler Interlocking Flag*, p.82, numerous packet-complete interrupts are a detriment to performance.

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

- When the send routine calls the transmit descriptor clean routine, the device may not have actually transmitted the packet and there may be little or nothing to clean. Therefore, the descriptor cleanup often depends on subsequent calls to the send routine to clean up previously used descriptors.
- Calling the transmit descriptor clean routine for every packet sent imposes substantial overhead.
In some circumstances, the first consideration can result in a transmit stall or even deadlock. The solution to this transmit stall is to continue to allow the packet-complete interrupt to occur but control the frequency at which it is generated. This gives a backup to the send routine’s cleanup attempts.

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

To determine if sufficient descriptors have been freed:
- Establish a threshold of some percentage of the transmit descriptors
- If the send routine’s call to the transmit descriptor clean routine does not increase the free count to greater than the threshold amount, unmask the packet-complete interrupt

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

Now put these two mitigators together:
- Only call the transmit descriptor clean routine when the free transmit descriptor count falls below a certain threshold.
- If the send routine’s call to the transmit descriptor clean routine does not increase the free count to a value greater than the given threshold, unmask the packet-complete interrupt.

Transmit Descriptor Indexing

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

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

free queue
  Lists descriptors currently available for use.

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

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

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

**Transmit Packet Association List**

As stated previously, it is the responsibility of the driver’s send to store a transmitted packet’s mBlk chain pointer in such a way that it can be later correlated to the associated descriptor or descriptors on the transmit queue. The mechanism to do this is a transmit packet association list. This list is an array of mBlk pointers that is of equal length to the total number of transmit descriptors allocated by the driver. This list is accessed using the same indices that the driver uses to reference the descriptors. When the send routine places a descriptor on the device transmit queue, it uses the free index to correlate the transmit packet association list to the transmit descriptor ring. As the send routine moves around the transmit descriptor ring, for each fragment buffer pointer it puts into a descriptor, it determines if that fragment is the last fragment for the packet it is transmitting. If it is the last fragment for the packet, the send routine puts the pointer to the packet’s mBlk chain into the transmit packet association list at the same index as the descriptor that holds the packet’s last fragment. If the fragment is not the last fragment of packet, the send routine sets the correlating transmit packet association list entry to NULL.

**Transmit-Packet-Complete Handler**

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

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

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

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

Transmit Descriptor Clean

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

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

TCP/IP checksum offloading eliminates host-side checksum computation with hardware assist. Many devices provide support for this feature.

The device and the host-side driver must act in concert to implement checksum offloading. The device supports checksum offloading in the DMA engine. The DMA engine computes the raw 16-bit ones-complement checksum of each DMA transfer as it moves the data to and from host memory. Using this checksum requires setting CSUM flags in the packet’s mBlk to either bypass the software checksum computation for received packets, or to alert the device that it needs to compute and insert checksums before transmitting a frame.

TCP or UDP checksumming actually involves two checksums—one for the IP header (including fields overlapping with the TCP or UDP header) and a second end-to-end checksum covering the TCP or UDP header and packet data. In a conventional system, TCP or UDP computes its end-to-end checksum before IP fills in its overlapping IP header fields (for example, options) on the sender, and after the IP layer restores these fields on the receiver. Checksum offloading involves computing these checksums below the IP stack; thus, the driver or device firmware must partially dismantle the IP header in order to compute a correct checksum. Instead of computing the checksum over the actual data fields of the TCP segment only, a 12-byte TCP pseudo header is created prior to checksum calculation. This header contains important information taken from fields in the TCP header, as well as the IP header into which the TCP segment is encapsulated. These fields include:

**source address**
- The 32-bit IP address of the originator of the datagram, taken from the IP header.

**destination address**
- The 32-bit IP address of the intended recipient of the datagram, also from the IP header.

**reserved**
- 8 bits of zeroes.

**protocol**
- The protocol field from the IP header. This indicates what higher-layer protocol is carried in the IP datagram. The protocol, TCP, is already known so this field normally has a value of 6.
TCP length
The length of the TCP segment, including both header and data.

NOTE: The TCP length is not a specified field in the TCP header, but is computed.

The Wind River checksum offloading API consists of the END_CAPABILITIES structure defined in end.h, the csum_flags and csum_data fields in the mBlkPktHdr structure in the mBlk, and the EIOCGIFCAP and EIOCSIFCAP ioctls in the driver.

The driver configures the device’s hardware registers to enable checksum offloading. The driver then initializes the END_CAPABILITIES structure with the capabilities that the device supports and has enabled. The fields of the END_CAPABILITIES structure hold the capabilities available, those currently enabled, and the CSUM flags for receive and transmit.

```c
typedef struct _END_CAPABILITIES {
    uint64_t cap_available; /* supported capabilities (RO) */
    uint64_t cap_enabled; /* subset of above which are enabled (RW) */
    uint32_t csum_flags_tx; /* cap_enabled mapped to CSUM flags for TX (RO) */
    uint32_t csum_flags_rx; /* cap_enabled mapped to CSUM flags for RX (RO) */
} END_CAPABILITIES;
```

The `cap_available` field reflects the capabilities supported by the driver. The `cap_enabled` field reflects the capabilities supported by the network stack. The driver loads the `cap_available` field with the capabilities supported by the device and initializes the `cap_enabled` field with the same values. Later, the network stack uses the driver’s ioctl to determine what capabilities the driver supports. The network stack may then change the `cap_enabled` field to request capabilities that it supports. It is not an error if the stack requests `cap_enabled` capabilities that the driver does not have available. However, the capabilities are not provided.

The `csum_flags_tx` and `csum_flags_rx` fields contain translations of `cap_available` and `cap_enabled` into CSUM flags. The CSUM flags provide more detailed information about the particular operations supported.

The END_CAPABILITIES structure is initialized in the driver’s `endLoad()` routine. The driver uses the capability flags defined in end.h to initialize the `cap_available` and `cap_enabled` fields and the CSUM flags defined in mbuf.h to initialize the `csum_flags_tx` and `csum_flags_rx` fields.

For example, if the network stack requests transmit checksum support by setting IFCAP_TXCSUM in `cap_enabled` and the `cap_available` field reflects that the driver
supports transmit checksumming by also having the IFCAP_TXCSUM bit set. The driver might set the csum_flags_tx field as follows:

\[(\text{CSUM\_IP} | \text{CSUM\_TCP} | \text{CSUM\_UDP})\]

Interface capabilities flags for the cap_available and cap_enabled fields are listed in Table 4-2.

### Table 4-2 Interface Capability Flags for cap_available and cap_enabled

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFCAP_RXCSUM</td>
<td>Supports IPv4 receive checksum offload</td>
</tr>
<tr>
<td>IFCAP_TXCSUM</td>
<td>Supports IPv4 transmit checksum offload</td>
</tr>
<tr>
<td>IFCAP_NETCONS</td>
<td>Supports being a network console</td>
</tr>
<tr>
<td>IFCAP_VLAN_MTU</td>
<td>Supports VLAN-compatible MTU</td>
</tr>
<tr>
<td>IFCAP_VLAN_HWTAGGING</td>
<td>Supports hardware VLAN tags</td>
</tr>
<tr>
<td>IFCAP_JUMBO_MTU</td>
<td>Supports 9000 byte MTU</td>
</tr>
<tr>
<td>IFCAP_TCPSEG</td>
<td>Supports IPv4/TCP segmentation</td>
</tr>
<tr>
<td>IFCAP_IPSEC</td>
<td>Supports IPsec</td>
</tr>
<tr>
<td>IFCAP_RXCSUMv6</td>
<td>Supports IPv6 receive checksum offload</td>
</tr>
<tr>
<td>IFCAP_TXCSUMv6</td>
<td>Supports IPv6 transmit checksum offload</td>
</tr>
<tr>
<td>IFCAP_TCPSEGv6</td>
<td>Supports IPv6/TCP segmentation</td>
</tr>
<tr>
<td>IFCAP_IPCOMP</td>
<td>Supports IPcomp</td>
</tr>
<tr>
<td>IFCAP_CAP0</td>
<td>Vendor specific capability #0</td>
</tr>
<tr>
<td>IFCAP_CAP1</td>
<td>Vendor specific capability #1</td>
</tr>
<tr>
<td>IFCAP_CAP2</td>
<td>Vendor specific capability #2</td>
</tr>
</tbody>
</table>

Flags indicating hardware checksum support and software checksum requirements are listed in Table 4-3.
Table 4-3  Hardware and Software Checksum Support Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSUM_IP</td>
<td>Enable IP checksum</td>
</tr>
<tr>
<td>CSUM_TCP</td>
<td>Enable TCP checksum</td>
</tr>
<tr>
<td>CSUM_UDP</td>
<td>Enable UDP checksum</td>
</tr>
<tr>
<td>CSUM_IP_FRAGS</td>
<td>Enable checksum of IP fragments (currently not supported)</td>
</tr>
<tr>
<td>CSUM_FRAGMENT</td>
<td>Stack can fragment IP packets</td>
</tr>
<tr>
<td>CSUM_TCP_SEG</td>
<td>Stack can segment TCP/IPv4</td>
</tr>
<tr>
<td>CSUM_TCPv6</td>
<td>Enable checksum for TCP/IPv6</td>
</tr>
<tr>
<td>CSUM_UDPV6</td>
<td>Enable checksum for UDP/IPv6</td>
</tr>
<tr>
<td>CSUM_TCPv6_SEG</td>
<td>Stack can segment TCP/IPv6</td>
</tr>
<tr>
<td>CSUM_IP_CHECKED</td>
<td>IP checksum done</td>
</tr>
<tr>
<td>CSUM_IP_VALID</td>
<td>IP checksum is valid</td>
</tr>
<tr>
<td>CSUM_DATA_VALID</td>
<td>csum_data field is valid</td>
</tr>
<tr>
<td>CSUM_PSEUDO_HDR</td>
<td>csum_data has pseudo header</td>
</tr>
<tr>
<td>CSUM_DELAY_DATA</td>
<td>(CSUM_TCP</td>
</tr>
<tr>
<td>CSUM_DELAY_IP</td>
<td>(CSUM_IP)</td>
</tr>
<tr>
<td>CSUM_DELAY_DATA6</td>
<td>(CSUM_TCPv6</td>
</tr>
<tr>
<td>CSUM_RESULTS</td>
<td>(CSUM_IP_CHECKED</td>
</tr>
<tr>
<td>CSUM_IP_HDRLEN</td>
<td>(pMblk) ((pMblk)-&gt;mBlkHdr.offset1)</td>
</tr>
<tr>
<td></td>
<td>Used to determine the actual IP header length</td>
</tr>
<tr>
<td>CSUM_XPORT_HDRLEN</td>
<td>(pMblk) (((pMblk)-&gt;mBlkPktHdr.csum_data &amp; 0xff00) &gt;&gt; 8)</td>
</tr>
<tr>
<td></td>
<td>Location of TCP or UDP checksum field</td>
</tr>
<tr>
<td>CSUM_XPORT_CSUM_OFF</td>
<td>(pMblk) ((pMblk)-&gt;mBlkPktHdr.csum_data) &amp;</td>
</tr>
</tbody>
</table>
The **END_CAPABILITIES** structure is used by the driver’s interface capabilities set and get ioctls:

**EIOCSIFCAP**
- Interface capabilities set ioctl.

**EIOCGIFCAP**
- Interface capabilities get ioctl.

For **EIOCGIFCAP**, the driver returns all fields according to its current settings. The stack need not initialize any of the fields.

**EIOCGIFCAP Example:**

```c
case EIOCGIFCAP:
    hwCaps = (END_CAPABILITIES *)data;

    if (hwCaps == NULL)
    {
        error = EINVAL;
        break;
    }
    hwCaps->csum_flags_tx = pDrvCtrl->hwCaps.csum_flags_tx;
    hwCaps->csum_flags_rx = pDrvCtrl->hwCaps.csum_flags_rx;
    hwCaps->cap_available = pDrvCtrl->hwCaps.cap_available;
    hwCaps->cap_enabled = pDrvCtrl->hwCaps.cap_enabled;
    break;
```

For **EIOCSIFCAP**, the stack sets the capabilities that it wants enabled into **cap_enabled**. This allows the stack to turn capabilities on or off as required. The stack can request any capability for which it is itself capable. If the stack requests capabilities that are not supported by the device, it is not an error. However, the driver only allows those capabilities that are set in the **cap_available** field, all other capabilities are ignored.

**EIOCSIFCAP Example:**

```c
case EIOCSIFCAP:
    hwCaps = (END_CAPABILITIES *)data;

    if (hwCaps == NULL)
    {
        error = EINVAL;
        break;
    }
    pDrvCtrl->hwCaps.cap_enabled = hwCaps->cap_enabled;
    break;
```
Checksum Offloading and Receiving

The driver’s receive routine:

1. Checks if the network stack has requested that the device-calculated checksum be passed to the stack. This is accomplished by testing to see if IFCAP_RXCSUM is set in the cap_enabled field in driver’s copy of the END_CAPABILITIES structure.

2. If receive checksumming is enabled, the driver reads the device’s checksum status register.

   a. The driver determines if the device calculated the IP checksum
      i. If the device calculated the IP header checksum, the driver sets CSUM_IP_CHECKED in the packet mBlk->mBlkPktHdr.csum_flags to indicate that the IP header checksum was calculated.

   b. The driver tests to see if the device determined that the IP header is valid.
      i. If the IP header is valid, the driver sets CSUM_IP_VALID in the packet mBlk->mBlkPktHdr.csum_flags to indicate that the IP header is valid.

   c. The driver tests if the device calculated the TCP or UDP checksum. It also tests to see that the checksum is valid, which indicates that the packet is uncorrupted.
      i. If the device calculated the TCP or UDP checksum and determined that the packet is valid, the driver sets CSUM_DATA_VALID in the packet mBlk->mBlkPktHdr.csum_flags to indicate that the TCP or UDP checksum was calculated and that the packet is valid.

      ii. If the device also computed the pseudo header, the driver sets CSUM_PSEUDO_HDR in the packet mBlk->mBlkPktHdr.csum_flags to indicate that the pseudo header was computed.

      iii. If the driver determines that the packet and checksum are valid, it writes the checksum into the mBlk at pMblk->m_pkthdr.csum_data. It is unnecessary to read the calculated checksum from a device register. A valid checksum is 0xffff, it is only necessary to write this value into the mBlk.
Handling Corrupt Packets

If a packet is corrupted, the driver has two options. It can choose to not set the CSUM flags in the mBlk or it can insert an invalid checksum value into the mBlk. In either case, the network stack recalculates the checksum. For example:

```c
/* Do RX checksum offload, if enabled. */
if (pDrvCtrl->hwCaps.cap_enabled & IFCAP_RXCSUM)
{
    /* Read the device checksum status register */
    RFD_BYTE_RD (pRbdTag->pRFD, RFD_CSRSTS_OFFSET, csumStatus);

    /* Determine if IP checksum calculated */
    if (csumStatus & RFD_CS_IP_CHECKSUM_BIT_VALID)
    {
        /* Set mBlk checksum flags to indicate checksum calculated */
        pRbdTag->pMblk->m_pkthdr.csum_flags |= CSUM_IP_CHECKED;
    }

    /* Determine if IP checksum valid */
    if (csumStatus & RFD_CS_IP_CHECKSUM_VALID)
    {
        /* Set mBlk checksum flags to indicate a valid IP header */
        pRbdTag->pMblk->m_pkthdr.csum_flags |= CSUM_IP_VALID;
    }

    if (csumStatus & RFD_CS_TCPUDP_CHECKSUM_BIT_VALID &&
        csumStatus & RFD_CS_TCPUDP_CHECKSUM_VALID)
    {
        pRbdTag->pMblk->m_pkthdr.csum_flags =
            CSUM_DATA_VALID|CSUM_PSEUDO_HDR;
        pRbdTag->pMblk->m_pkthdr.csum_data = 0xFFFF;
    }
}
```

Checksum Offloading and Transition

The stack always computes the pseudo header. The device can overwrite it but the stack always calculates it. This cannot be turned off.

The network stack communicates to the driver about whether or not to instruct the device to calculate a checksum for a given packet through CSUM flags in pMblk->m_pkthdr.csum_flags.

The checksums are stored in the headers at the front of each IP packet, the device must complete the checksum before it can transmit the packet headers. Because the checksums are computed by the device’s DMA engine, the last byte of the packet must arrive in the device before it can determine the complete checksum. That is, in order for the device to calculate a checksum on a packet, it must delay transmission of any part of the packet until after it has processed the entire packet.
The driver’s send routine must:

1. Determine whether or not the network stack needs the device to calculate checksums for the packet it is processing. To do this, the routine reads `pMblk->m_pkthdr.csum_flags`.
   a. If the network stack requests that the device calculate the IP checksum, the driver prepares to set the device accordingly.
   b. If the network stack requests that the device calculate the TCP or UDP checksum, the driver prepares to set the device accordingly.

2. After the driver interprets the `CSUM` flags and prepares to set the device accordingly, it writes the appropriate settings into the device’s register.

For example:

```c
/* Do TX checksum offload. */
if (pDrvCtrl->csumOffload)
{
    txCsum = 0;
    if (pMblkHead->m_pkthdr.csum_flags)
    {
        txCsum = (IPCB_HARDWAREPARSING_ENABLE << 8);
        if (pMblkHead->m_pkthdr.csum_flags & CSUM_IP)
            txCsum |= IPCB_IP_CHECKSUM_ENABLE;
        if (pMblkHead->m_pkthdr.csum_flags & CSUM_TCP)
            txCsum |= IPCB_TCP_PACKET;
    }
    CFD_WORD_WR (pCFD, CFD_IPSCHED_OFFSET, txCsum);
}
```

Implementing Required Entry Points and Structures

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

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

Within your driver, you must allocate and initialize an END_OBJ. Your driver also needs to allocate and initialize the structures referenced in END_OBJ structures, such as DEV_OBJ, NET_FUNCS, and M2_INTERFACE_TBL. To pass packets up to the MUX, use an mBlk structure.

Providing Network Device Abstraction: END_OBJ

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

typedef struct end_object
{
  NODE node; /* root of the device hierarchy */
  DEV_OBJ devObject; /* accesses your device’s ctrl struct */
  FUNCPTR receiveRtn; /* routine to call on reception */
  BOOL attached; /* indicates unit is attached */
  SEM_ID txSem; /* transmitter semaphore */
  long flags; /* various flags */
  struct net_funcs *pFuncTable; /* function table */
  M2_INTERFACE_TBL mib2Tbl; /* MIBII counters */
  struct ETHER_MULTI *pAddrList; /* head of the multicast address list */
  int nMulti; /* number of elements in the list */
  LIST protocols; /* protocol node list */
  BOOL snarfProto; /* is someone snarfing us? */
  void* pMemPool; /* memory cookie used by MUX bufr mgr. */
  M2_ID* pMib2Tbl; /* RFC 2233 MIB objects */
} END_OBJ;

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

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

devObject
The DEV_OBJ structure for this device. Your driver must set this value at load time. See Tracking Your Device’s Control Structure: DEV_OBJ, p.99.

receiveRtn
A function pointer that references a muxReceive() routine. The MUX supplies this pointer by the completion of the muxDevLoad() call that loads this driver. Your driver uses this function pointer to pass data up to the protocol.
attached
A BOOL indicating whether or not the device is attached. The MUX sets and manages this value.

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

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

  IFF_UP
  The interface driver is up.

  IFF.Broadcast
  The broadcast address is valid.

  IFF_DEBUG
  Debugging is on.

  IFF_LOOPBACK
  This is a loopback net.

  IFF.POINTOPOINT
  The interface is a point-to-point link.

  IFF.NOTRAILERS
  The device must avoid using trailers.

  IFF.RUNNING
  The device has successfully allocated needed resources.

  IFF_NOARP
  There is no address resolution protocol.

  IFF.PROMISC
  This device receives all packets.

  IFF_ALLMULTI
  This device receives all multicast packets.

  IFF_OACTIVE
  Transmission in progress.

  IFF_SIMPLEX
  The device cannot hear its own transmissions.
**IFF_LINK0, IFF_LINK1, IFF_LINK2**  
Per link layer defined bits.

**IFF_MULTICAST**  
The device supports multicast.

**IFF_LOAN**  
The device supports buffer loaning.

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

**pFuncTable**  
A pointer to a `NET_FUNCS` structure. This structure contains function pointers to your driver’s entry points for handling standard requests such as unload or send. Your driver must allocate and initialize this structure when the device is loaded. See *Identifying the Entry Points into Your Network Driver: NET_FUNCS*, p.100.

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

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

**pAddrList**  
A pointer to the head of a list of multicast addresses. The MUX sets and manages this list, but it uses your driver’s `endMCastAddrAdd()`, `endMCastAddrDel()`, and `endMCastAddrGet()` entry points to do so.

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

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

**snarfProto**  
A `BOOL` indicating whether a packet-snarfing protocol has bound itself to this driver. Such a protocol can prevent the packet from passing on to lower
priority protocols (see Protocol Startup, p.44). The MUX sets and manages this value.

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

pMib2Tbl
The interface table for RFC 2233 compliance.

**Tracking Your Device’s Control Structure: DEV_OBJ**

Your driver uses the DEV_OBJ structure to tell the MUX the name of your device and to hand the MUX a pointer to your device’s control structure. This control structure is a device-specific structure that you define according to your needs. Your driver uses this control structure to track things such as flags, memory pool addresses, and so on. The information stored in the control structure is typically essential to just about every driver entry point. The DEV_OBJ structure is defined in end.h as follows:

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

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

**pDevice**
A pointer to your driver’s internal control structure. This field was originally intended as a back pointer to the driver control structure. The driver used this field to dereference itself from the pCookie passed from MUX calls. However, in a properly initialized END driver, this field is NULL. This is because an END driver should pass the END_OBJ_INIT macro NULL as the pDevice argument. The reason for this is that passing the device’s control structure pointer results in the MUX freeing the structure when the device is unloaded from the MUX. Because the driver stores other ancillary pointers in its control structure (which it cannot free until after it has been unloaded from the MUX), it must preserve this pointer. The pointer is preserved by passing the NULL as pDevice in END_OBJ_INIT. Therefore, this field is deprecated and should not be used unless a driver allocates END_OBJ separately from its control structure (this practice is not recommended).
unit
This is the unit number for the particular named device. Unit numbers start at 0 and increase for every device controlled by the same driver. For example, if a system has two Lance Ethernet devices (named ln) then the first one is ln0 and the second is ln1. If the same system also has a DEC 21x40 Ethernet, that device (whose name is dc) is dc0.

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

Identifying the Entry Points into Your Network Driver: NET_FUNCS
The MUX uses the NET_FUNCS structure to maintain a table of entry points into your END driver. The NET_FUNCS structure is defined as follows:

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

Within your endLoad() routine, initialize these members to point to the appropriate driver entry points. Thus, start should contain a pointer to your endStart(), stop to your endStop(), unload to your endUnload(), and so on.

Tracking Link-Level Information: LL_HDR_INFO
The MUX uses LL_HDR_INFO structures to keep track of link-level header information associated with packets passed from an END driver to the MUX and from there up to a protocol. An LL_HDR_INFO structure is passed as an argument to all stack receive routines (see, Passing a Packet Up to the Protocol: stackRcvRtn(), p.49).
typedef struct llHdrInfo
{
  int destAddrOffset; /* destination addr offset in mBlk */
  int destSize;      /* destination address size */
  int srcAddrOffset; /* source address offset in mBlk */
  int srcSize;       /* source address size */
  int ctrlAddrOffset; /* control info offset in mBlk */
  int ctrlSize;      /* control info size */
  int pktType;       /* type of the packet */
  int dataOffset;    /* data offset in the mBlk */
} LL_HDR_INFO;

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

destSize
  Size of destination address.

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

srcSize
  Size of source address.

ctrlAddrOffset
  Reserved for future use.

ctrlSize
  Reserved for future use.

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

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

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

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

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

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

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

pClBlk
Contains a pointer to a cBlk structure. You should have no need to access or set this member directly and can treat it as opaque. However, if you are not using netBufLib to manage the driver’s memory pool, you must provide your own memory free routine for its associated cluster. To do this, you must update mBlk.pClBlk.pClFreeRtn to point to your customized free routine. This routine must use the same API as the netBufLib free routine. This means that the mBlk.pClBlk.pFreeArg1, mBlk.pClBlk.pFreeArg2, and mBlk.pClBlk.pFreeArg3 members must also be updated.

Setting appropriate values for the members listed above (and the members of all the referenced structures) is just a matter of calling the appropriate netBufLib routines for the creation of an mBlk/cBlk/cluster construct (or tuple). For more information, see Setting Up and Using Memory for Receive and Transmit Buffers, p.64.

Required Driver Entry Points

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

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>endLoad()</code></td>
<td>Initialize the driver and load it into the MUX.</td>
</tr>
<tr>
<td><code>endUnload()</code></td>
<td>Free driver resources.</td>
</tr>
<tr>
<td><code>endStart()</code></td>
<td>Start the driver.</td>
</tr>
<tr>
<td><code>endStop()</code></td>
<td>Stop the driver.</td>
</tr>
<tr>
<td><code>endSend()</code></td>
<td>Send a packet out on the hardware.</td>
</tr>
<tr>
<td><code>endIoctl()</code></td>
<td>Access driver control routines.</td>
</tr>
<tr>
<td><code>endMCastAddrAdd()</code></td>
<td>Add an address to the device’s multicast address list.</td>
</tr>
<tr>
<td><code>endMCastAddrDel()</code></td>
<td>Delete an address from the device’s multicast address list.</td>
</tr>
<tr>
<td><code>endMCastAddrGet()</code></td>
<td>Get the list of multicast addresses maintained for this device.</td>
</tr>
<tr>
<td><code>endPollSend()</code></td>
<td>Do a polling send.</td>
</tr>
<tr>
<td><code>endPollReceive()</code></td>
<td>Do a polling receive.</td>
</tr>
<tr>
<td><code>endAddressForm()</code></td>
<td>Add the appropriate link-level information into an <code>mBlk</code> in preparation for transmission. This routine is provided by the network stack and not typically defined by the driver.</td>
</tr>
</tbody>
</table>
Loading the Device: endLoad()

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

Your endLoad() routine must take the following form:

```c
END_OBJ* endLoad
{
  char* initString /* a string encoded for the device to use for its */
  /* initialization arguments. */
}
```

Within the endLoad() routine, you must handle any device-specific initialization. You should also set values for most of the members of the END_OBJ structure. Of particular interest are the END_OBJ members receiveRtn, pFuncTable, and devObject. For more information on these members, see the member descriptions provided in Providing Network Device Abstraction: END_OBJ, p.96.

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

---

Table 4-4 Required Driver Entry Points (cont’d)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>endPacketDataGet()</code></td>
<td>Extract packet data (omitting link-level information) from one mBlk and write it to another. This routine is provided by the network stack and not typically defined by the driver.</td>
</tr>
<tr>
<td><code>endPacketAddrGet()</code></td>
<td>Extract address information (omitting packet data) from one mBlk and write out each source and destination address to its own mBlk. For an Ethernet packet, this requires two output mBlks. However, for some non-Ethernet packets, this could require as many as four output mBlks because the local source and destination addresses can differ from the ultimate source and destination addresses. This routine is provided by the network stack and not typically defined by the driver.</td>
</tr>
</tbody>
</table>

---

External Interface

---

PacketDataGet() Extract packet data (omitting link-level information) from one mBlk and write it to another. This routine is provided by the network stack and not typically defined by the driver.

PacketAddrGet() Extract address information (omitting packet data) from one mBlk and write out each source and destination address to its own mBlk. For an Ethernet packet, this requires two output mBlks. However, for some non-Ethernet packets, this could require as many as four output mBlks because the local source and destination addresses can differ from the ultimate source and destination addresses. This routine is provided by the network stack and not typically defined by the driver.
The argument is:

**initString**

Passes in any initialization parameters needed.

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

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

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

```c
END_OBJ * templateEndLoad
(
    char *initString /* parameter string */
)
{
    DRV_CTRL * pDrvCtrl; /* pointer to DRV_CTRL structure */
    ...
    if (initString == NULL)
        return (NULL);
    if (initString[0] == 0)
    {
        bcopy ((char *)DEV_NAME, (void *)initString, DEV_NAME_LEN);
        return (0);
    }
}
```

*endLoad()* configures the device’s registers to either the default values or as prescribed by the driver parameters.

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

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

Additional routines are necessary for network stack operations. Entry points to
these routines are provided by the NET_FUNCS structure, which is pointed to by
an entry in the END_OBJ structure. Normally, these routines are declared local to
the driver and are only accessed through the NET_FUNCS structure. For a
description of the driver routines, see Table 4-4.

**Unloading the Device: endUnload()**

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

endUnload() must take the following form:

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

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

The parameters are:

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

- **Unloading an END Driver**

  NOTE: This example assumes a VxWorks 6.x environment and the use of
  netPoolCreate() to establish the driver’s buffer pool. Also, to use these
  instructions, an END driver must pass a NULL as the second argument to
  END_OBJ_INIT.
The unload routine in an END driver can only be called through `muxUnload()`. Before the `muxUnload()` routine calls the driver’s unload routine, it must unbind the device driver from any protocols to which it was previously bound. The driver’s unload routine must then complete the unload by:

- disabling the device
- freeing its associated memory

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

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

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

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

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

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

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

- there is no possibility of the memory being inadvertently accessed
- the memory is not holding the only copy of a pointer to allocated memory
These conditions impose a specific sequence of events for freeing the memory areas:

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

\textbf{NOTE:} The macro call to \texttt{END_OBJ_INIT} must have a NULL as its second argument. Otherwise, the MUX attempts to free the driver’s control structure resulting in a double free error.

9. Exit the unload routine.

\textbf{Providing an Opaque Control Interface to Your Driver: endIoctl()}

Your \texttt{endIoctl() }entry point should handle all requests for changes to the state of the device, such as bringing it up, shutting it down, turning on promiscuous mode, and so on. You can also use your \texttt{endIoctl() }routine to provide access to MIB-II interface statistics.
Your `endioctl()` must take the following form:

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

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

The parameters are:

- **pCookie**
  Passes a pointer to the `END_OBJ` structure returned by `endLoad()`.

- **cmd**
  Can pass any of the values shown in the command column of Table 4-5. Your `endioctl()` must have an appropriate response to each command.

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

### Table 4-5: `ioctl` Commands and Data Types

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIOCSFLAGS</td>
<td>Set device flags.</td>
<td>int; see description of <code>END_OBJ.flags</code></td>
</tr>
<tr>
<td>EIOCGFLAGS</td>
<td>Get device flags.</td>
<td>int</td>
</tr>
<tr>
<td>EIOCSADDR</td>
<td>Set device address.</td>
<td>char*</td>
</tr>
<tr>
<td>EIOCGADDR</td>
<td>Get device address.</td>
<td>char*</td>
</tr>
<tr>
<td>EIOCMULTIADD</td>
<td>Add multicast address.</td>
<td>char*</td>
</tr>
<tr>
<td>EIOCMULTIDEL</td>
<td>Delete multicast address.</td>
<td>char*</td>
</tr>
<tr>
<td>EIOCMULTIGET</td>
<td>Get multicast list.</td>
<td>MULTI_TABLE*</td>
</tr>
<tr>
<td>EIOCPOLLSTART</td>
<td>Set device into polling mode.</td>
<td>NULL</td>
</tr>
<tr>
<td>EIOCPOLLSTOP</td>
<td>Set device into interrupt mode.</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Sending Data Out on the Device: endSend()

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

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

This routine should return OK, ERROR, or END_ERR_BLOCK.

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

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

The parameters are:

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

---

**Table 4-5 Ioctl Commands and Data Types**

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIOCGPOLLCONF</td>
<td>Configure a data location from which the network stack can read statistics</td>
<td>END_IFDRVCONF*</td>
</tr>
<tr>
<td>EIOCGPOLLSTATS</td>
<td>Return network statistics to the caller</td>
<td>END_IFCOUNTERS*</td>
</tr>
<tr>
<td>EIOCGFBUF</td>
<td>Get minimum first buffer for chaining.</td>
<td>int</td>
</tr>
<tr>
<td>EIOCGMIB2</td>
<td>Get the MIB-II counters from the driver.</td>
<td>M2_INTERFACETBL*</td>
</tr>
</tbody>
</table>

---

- EIOCGPOLLCONF
- EIOCGPOLLSTATS
- EIOCGFBUF
- EIOCGMIB2

---

Table 4-5 Ioctl Commands and Data Types

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIOCGPOLLCONF</td>
<td>Configure a data location from which the network stack can read statistics</td>
<td>END_IFDRVCONF*</td>
</tr>
<tr>
<td>EIOCGPOLLSTATS</td>
<td>Return network statistics to the caller</td>
<td>END_IFCOUNTERS*</td>
</tr>
<tr>
<td>EIOCGFBUF</td>
<td>Get minimum first buffer for chaining.</td>
<td>int</td>
</tr>
<tr>
<td>EIOCGMIB2</td>
<td>Get the MIB-II counters from the driver.</td>
<td>M2_INTERFACETBL*</td>
</tr>
</tbody>
</table>

---

- EIOCGPOLLCONF
- EIOCGPOLLSTATS
- EIOCGFBUF
- EIOCGMIB2

---

Sending Data Out on the Device: endSend()

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

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

This routine should return OK, ERROR, or END_ERR_BLOCK.

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

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

The parameters are:

- **pCookie**
  
  Passes a pointer to the END_OBJ structure returned by endLoad(). Because the first field in the driver’s control structure (DRV_CTRL) is always END_OBJ, most drivers expect pDrvCtrl. This is allowed because pCookie and pDrvCtrl are interchangeable.
pMblk
   Passes a pointer to an mBlk structure containing the data you want to send. For more information on how to setup an mBlk, see Setting Up and Using Memory for Receive and Transmit Buffers, p. 64.
   In most cases, a transmit-done interrupt routine schedules a task-level routine to free the mBlk after the packet is sent.

Starting a Stopped but Loaded Driver: endStart()
   Your endStart() entry point should do whatever is necessary to make the driver active. For example, it should register your device driver’s interrupt service routine. Your endStart() routine must take the following form:

   Status endStart
   (    void* pCookie /* pointer to device-identifying END_OBJ structure */    )
   This routine should return OK or ERROR. If an error occurs, the routine should set errno.
   The parameters are:

   pCookie
   Passes a pointer to the END_OBJ structure returned by endLoad(). Because the first field in the driver’s control structure (DRV_CTRL) is always END_OBJ, most drivers expect pDrvCtrl. This is allowed because pCookie and pDrvCtrl are interchangeable.
   However, your endStart() should probably include this pointer as a parameter to the sysIntConnect() routine that it uses to register the ISR. The ISR may not have any direct use for the END_OBJ pointer, but it should pass the pointer into the driver entry point that handles task-level processing for packet reception.
   When it comes time to pass the packet up to the MUX, your driver must call the MUX-supplied routine referenced in pCookie.receiveRtn. See Providing Network Device Abstraction: END_OBJ, p. 96.

Stopping the Driver Without Unloading It: endStop()
   Your endStop() entry point can assume that the driver is already loaded and that endLoad() has already been called. Within your endStop() routine, you should do whatever is necessary to make the driver inactive without actually unloading the driver. endStop() must take the following form:
This routine should return OK or ERROR. If an error occurs, the routine should set errno.

The parameters are:

pCookie

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

Handling a Polled Send: endPollSend()

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

dPollSend() must take the following form:

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

Within your endPollSend() routine, check that the device is set to polled mode (by a previous endIoctl() call). Wind River recommends that your endPollSend() routine keep a transmit tuple, allocated from the driver’s pool, permanently available for its use. The pointer to this tuple should be stored in driver’s DRV_CTRL structure.

```c
if ((pDrvCtrl->pTxPollMblk = netTupleGet (pDrvCtrl->endObj.pNetPool,
 ETHERMTU + /* max data portion */
 16 + /* size of enet header */
 4, /* FCS */
 M_DONTWAIT, MT_DATA, FALSE)) == NULL)
{
 pDrvCtrl->lastError.errCode = END_ERR_NO_BUF;
 muxError(&pDrvCtrl->endObj, &pDrvCtrl->lastError);
 return ERROR;
}
```
Then, keep a pointer to the transmit tuple’s cluster buffer as follows:

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

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

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

The `endPollSend()` routine and the `endSend()` routine share the same transmit descriptors and the same transmit queue. Therefore, `endPollSend()` should treat the transmit queue and descriptors in the same manner as the `endSend()` routine.

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

The parameters are:

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

- `pMblk` 
  Passes a pointer to an `mBlk` structure containing the data you want to send. For information on setting up an `mBlk`, see Setting Up and Using Memory for Receive and Transmit Buffers, p.64.

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

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

`endPollReceive()` must take the following form:

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

Your `endPollReceive()` routine should check that the device is set to polled mode (by a previous `endIoctl()` call). Your `endPollReceive()` should then get a packet directly from the network and copy it to the `mBlk` passed in by the `pMblk` parameter.
Your `endPollReceive()` entry point should return OK or an appropriate error value. One likely error return value is `EAGAIN`. Your routine should return `EAGAIN` if the submitted `mBlk` is not big enough to contain the received packet, or if no packet is available.

The parameters are:

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

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

**Adding a Multicast Address: `endMCastAddrAdd()`**

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

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

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

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

The parameters are:

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

**pAddress**
Passes in a pointer to the address you want to add to the list. To help you manage a list of multicast addresses, VxWorks includes the library, `etherMultiLib`. 
Within your `endMCastAddrAdd()` routine, you must reconfigure the interface in a hardware-specific way. This reconfiguration should allow the driver to receive frames from the specified address and then pass those frames up to the higher layer.

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

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

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

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

The parameters are:

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

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

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

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

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

```c
STATUS endMCastAddrGet
{
    void* pCookie,
    MULTI_TABLE* pMultiTable
}
```
To get the list of multicast address, use the routines provided in `etherMultiLib`. This routine should return OK or ERROR. If an error occurs, the routine should set `errno`.

The parameters are:

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

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

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

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

**Forming an Address for Packet Transmission: `endAddressForm()`**

The `endAddressForm()` routine must take a source address and a destination address and copy the information into the data portion of the `mBlk` structure in a fashion appropriate to the link level. Implementing this functionality is the responsibility of the driver writer. However, a simple implementation of this routine is provided in `endLib`, you can use this routine as provided and are not required to provide your own. After adding the addresses to `mBlk`, your `endAddressForm()` routine should adjust the `mBlk.mBlkHdr.mLen` and `mBlk.mBlkHdr.mData` members accordingly. This routine must take the following form:

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

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

The parameters are:

- **pMblk**
  - The `mBlk` that contains the packet to be transmitted.
- **pSrcAddress**
  - The `mBlk` that contains the link-level address of the source.
- **pDstAddress**
  - The `mBlk` that contains the link-level address of the destination.

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

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

The routine is of the following form:

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

The parameters are:

- **pBuff**
  - Expects a pointer to the `mBlk` that contains both header and packet data.
- **pLinkHdrInfo**
  - Returns an `LL_HDR_INFO` structure containing header information that is dependent upon the particular data-link layer that the END driver implements. For more information, see *Tracking Link-Level Information: LL_HDR_INFO*, p.100.

### Return Addressing Information: endEtherPacketAddrGet( )

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

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

pSrc

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

pDst

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

pESrc

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

pEDst

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

## 4.3 The END Driver Development Process

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

### 4.3.1 Driver Development Overview

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

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

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

```c
typedef struct drv_ctrl
{
  END_OBJ   endObj;   /* base class */
  int       unit;     /* unit number */
  ...
} DRV_CTRL;
```

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

Buffer allocation is done during initialization. It is strongly recommended that netPoolCreate() be used as described in Setting Up a Memory Pool, p.66. Buffer allocation creates clusters, clBlks, and mBlks for transferring packets between the driver and the network stack. Both clBlks and mBlks are used by the driver and the network stack, but they are not handled by the device. Clusters are used by the network stack, the driver, and the device. For this reason, caching is an important concern. For more information on caching, see 3.3.2 Cache Considerations, p.26.

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

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

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

\begin{verbatim}
#ifndef TEMPLATE_BYTE_RD
#define TEMPLATE_BYTE_RD(addr, value) (value = sysInByte ((ULONG) addr))
#endif
\end{verbatim}

Additional low level code is used to manipulate the device structures. For more information on structures, see \textit{Implementing Required Entry Points and Structures}, p.95.

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

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

\textbf{Porting an Existing Driver From Another OS}

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

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

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

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

Additional Development Issues

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

Backwards Compatibility

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

Performance

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

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

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

4.3.2 Error Conditions

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

```c
void muxError
{
    void* pCookie, /* pointer to END_OBJ */
    END_ERR* pError /* pointer to END_ERR structure */
}
```

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

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

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

```c
void xxError
{
    END_OBJ* pEnd, /* pointer to END_OBJ */
    END_ERR* pError, /* pointer to END_ERR */
    void* pSpare /* pointer to protocol private data passed in muxBind */
}
```

The `errCode` member of an `end_err` structure is 32 bits long. Wind River reserves the lower 16 bits of `errCode` for its own error messages. However, the upper 16 bits are available to user applications. Use these bits to encode whatever error messages you need to pass between drivers and protocols. The currently defined error codes are as follows:
#define END_ERR_INFO 1 /* information only */
#define END_ERR_WARN 2 /* warning */
#define END_ERR_RESET 3 /* device has reset */
#define END_ERR_DOWN 4 /* device has gone down */
#define END_ERR_UP 5 /* device has come back on line */
#define END_ERR_FLAGS 6 /* device flags have changed */
#define END_ERR_NO_BUF 7 /* device's cluster pool is exhausted */

These error codes have the following meaning:

**END_ERR_INFO**
This error is information only.

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

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

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

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

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

**END_ERR_FLAGS**
The device flags have changed.

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

### 4.3.3 Generic MIB Interface Initialization

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

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

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

This document provides instructions for implementing both methods.

**Pushed Method**

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

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

**endM2Init()**

Drivers should call endM2Init() with the proper arguments in their endLoad() routine.

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

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

```c
endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd, [u_char *] &enetAddr[0], 6, ETHERMTU, speed, IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);
```
endM2ioctl()  
If a driver’s ioctl() is called with a EIOCGMIB or EIOCGMIB233, it must call endM2ioctl().

dendM2Packet()  
When a driver receives or sends a packet, encounters an error, or discards a packet, it must call endM2Packet().

endM2Packet(pEnd, pMBlk, counter)

Where counter is one of the following:

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

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

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

It is vital that all endM2Packet() calls be located in such a place that their validity is guaranteed. That is, do not log a successful receipt or send of a packet until it is absolutely certain that the packet has been successfully received or sent. Special care should be taken to ensure that all failure conditions are properly logged.

dendM2Free()  
A driver must call endM2Free() in its unload routine. This routine frees the appropriate structures (that is, any allocated by endM2Init()).

Implementing the Generic MIB Pushed Method

The following instructions document the process of implementing the generic MIB pushed method. With these instructions, you can convert an old RFC 1213 MIB interface to use the generic MIB or, you can use these instructions to implement the generic MIB in a driver that does not have the RFC 1213 MIB interface already implemented. If the original driver does not already support RFC 1213, ignore the instructions to remove the RFC 1213 interface API.
1. In the driver `endLoad()` routine, call `endM2Init()`.
    Initialize MIB-II entries (for RFC 2233 ifXTable)
    For example:
    ```c
    endM2Init(&pDrvCtrl->endObj, M2_ifType_ethernet_csmacd,
              (u_char *) &enetAddr[0], FEI_ADDR_LEN, ETHERMTU, speed,
              IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);
    ```

2. In the driver `endUnload()` routine, call `endM2Free()`.
    ```c
    endM2Free (pDrvCtrl);
    ```

3. Add the `EIOCGMIB2233` case in the `ioctl()` routine.
    If the driver's `ioctl()` is called with a `EIOCGMIB2` or `EIOCGMIB2233`, call `endM2Ioctl()`.
    For example:
    ```c
    /* New RFC 2233 mib2 interface */
    case EIOCGMIB2233:
        case EIOCGMIB2:
            endM2Ioctl (pDrvCtrl, cmd, datal);
            break;
    ```

4. Replace the old RFC 1213 interface API with the generic MIB interface API, then delete the old RFC 1213 interface API.
   RFC 1213 Interface API:
   The old RFC 1213 interface used the `END_ERR_ADD` macro for both updating packet counts and for counting error conditions.
   a. Replace all instances of `END_ERR_ADD` calls. After an `END_ERR_ADD` instance is replaced, it can be deleted.
      For example:
      ```c
      END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_UCAST, +1);
      ```
   b. Replace the deleted RFC 1213 Interface.
      i. In the send and polling send routines, add the generic `mib2` counter update for outgoing packets.
         Send routine:
         ```c
         endM2Packet(pDrvCtrl, pMBlk, M2_PACKET_OUT);
         ```
Polling send routine:

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

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

Receive routine:

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

Polling receive routine:

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

5. Log failure and error conditions.

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

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

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

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

Example:

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

**Pulled Method**

The following instructions detail the implementation of the generic MIB pulled method. It is not anticipated that these instructions will be used with drivers that have already implemented the RFC 1213 MIB interface.
1. Add `END_IFDRVCONF` and `END_IFCOUNTERS` structures to the driver's control structure as follows:

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

2. Declare a status dump routine.

   ```c
   LOCAL STATUS gei82543EndStatsDump (ENDDEVICE *);
   ```

3. Modify the END driver load routine.

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

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

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

4. Modify the `ioctl()` routine.

   ```c
   case EIOCGMIB2233:
   case EIOCGMIB2:
       endM2Ioctl (pDrvCtrl, cmd, datal);
       break;
   ```

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

5. Define an \texttt{xxxEndStatsDump()} routine.

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

In the following example, the device counts multicast and broadcast packets and all incoming packets but does not specifically count unicast packets. The \texttt{xxxEndStatsDump()} routine calculates the unicast value by subtracting the multicast and broadcast values from the count of all incoming packets.

Example:

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

    pEndStatsCounters = &pDrvCtrl->endStatsCounters;

    /*
    * Get number of RX'ed octets
    * Note: the octet counts are 64-bit quantities saved in two
    * 32-bit registers. Reading the high word clears the count,
    * so we have to read the low word first.
    */
```
GEI_READ_REG(INTEL_82543GC_GORL, tmp);
pEndStatsCounters->ifInOctets = tmp;
GEI_READ_REG(INTEL_82543GC_GORH, tmp);
pEndStatsCounters->ifInOctets |= (unsigned long long)tmp << 32;

/* Get number of TX'ed octets */
GEI_READ_REG(INTEL_82543GC_GOTL, tmp);
pEndStatsCounters->ifOutOctets = tmp;
GEI_READ_REG(INTEL_82543GC_GOTH, tmp);
pEndStatsCounters->ifOutOctets |= (unsigned long long)tmp << 32;

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

/* Get TX'ed unicasts, broadcasts, multicasts */
GEI_READ_REG(INTEL_82543GC_GPTC, tmp);
pEndStatsCounters->ifOutUcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_BPTC, tmp);
pEndStatsCounters->ifOutBroadcastPkts = tmp;
GEI_READ_REG(INTEL_82543GC_MPTC, tmp);
pEndStatsCounters->ifOutMulticastPkts = tmp;
pEndStatsCounters->ifOutUcastPkts -=
    (pEndStatsCounters->ifOutMulticastPkts +
     pEndStatsCounters->ifOutBroadcastPkts);

return (OK);
}

6. Modify the unload routine.

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

130
5.1 Introduction 131

5.2 BSD Ethernet Driver Overview 132

5.3 Upgrading Your Driver to 4.4 BSD 135

5.4 Porting a BSD Ethernet Driver to the END Model 139

5.1 Introduction

NOTE: BSD Ethernet drivers (also known as netif drivers) are not supported for VxWorks 6.x. The information in this chapter is applicable to VxWorks 5.5 only.

This chapter describes two port paths for 4.3 BSD network drivers. One path simply ports the 4.3 BSD network driver to the BSD 4.4 model. The other path ports the 4.3 BSD network driver to an END driver (described in 4. END Ethernet Drivers).

Porting a network driver to the 4.4 BSD model should require only minimal changes to the code. In fact, porting some drivers has taken less than a day’s work. However, an older driver that does not already use a transmission startup routine can take longer to port.
Porting a network driver to an END requires more extensive changes. However, it is worth the effort if the driver must handle the following:

- multicasting
- polled-mode Ethernet (necessary for WDB debugging of the kernel over a network). This mode is several orders of magnitude faster than the serial link.
- zero-copy transmission
- support for network protocols other than IP

NOTE: This chapter assumes that you are already familiar with BSD network device drivers. You should also be familiar with 4. END Ethernet Drivers.

5.2 BSD Ethernet Driver Overview

This section provides an overview of BSD Ethernet driver structure. It also includes a discussion of etherhooks.

5.2.1 Structure of a 4.3 BSD Network Driver

This section describes netif drivers for VxWorks that are based on those available in BSD UNIX version 4.3. These drivers define only one global (user-callable) routine, the driver’s attach routine. Typically, the name of this routine contains the word, attach, prefixed with two letters from the device name. For example, the AMD Lance driver’s attach routine is called lnattach(). The xxattach() routine hooks in five function pointers that are mapped into an ifnet structure. These routines, listed in Table 5-1, are all called from various places in the IP protocol stack, which has intimate knowledge of the driver.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function Pointer</th>
<th>Driver-Specific Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>if_init</td>
<td>xxInit()</td>
</tr>
<tr>
<td>output</td>
<td>if_output</td>
<td>xxOutput()</td>
</tr>
</tbody>
</table>
Packet reception begins when the driver’s interrupt routine is invoked. The interrupt routine does the least work necessary to get the packet off the local hardware, schedules an input handler to run by calling `netJobAdd()`, and then returns. The `tNetTask` calls the routine that was added to its work queue. In the case of packet reception, this is typically the driver’s `xxReceive()` routine.

The `xxReceive()` routine eventually sends the packet up to a protocol by calling `do_protocol_with_type()`. This routine is a switch statement that figures out which protocol to hand the packet off to. This calling sequence is shown in Figure 5-1.

Figure 5-2 shows the call graph for packet transmission. After a protocol has picked an interface on which to send a packet, it calls the `xxOutput()` routine for that interface. The output routine calls the generic `ether_output()` routine, passing it a pointer to addressing information (usually an `arpcom` structure) as well as the data to be sent. After the data is properly packed, it is placed on the output queue.
(using the **IF_ENQUEUE** macro), and the driver’s start routine is called. The \texttt{\texttt{xxTxStartup}} routine de-queues as many packets as it can and transmits them on the physical medium.

**Figure 5-2** Packet Transmission Call Graph

![Packet Transmission Call Graph](image)

1. The \texttt{\texttt{xxTxStartup}} first shows the packet to \texttt{etherOutputHook()}.  
2. If \texttt{etherOutputHook()} does not take delivery of the packet, \texttt{\texttt{xxTxStartup}} transmits the packet on the medium.

### 5.2.2 Etherhook Routines Provide Access to Raw Packets

You can use the \texttt{etherInputHook()} and \texttt{etherOutputHook()} routines to bypass the TCP/IP stack and thus get access to raw packets. On packet reception, if an \texttt{etherInputHook()} routine is installed, it receives the packet just after the driver has completed reception but before the packet goes to the protocol. If \texttt{etherInputHook()} decides to prevent others from seeing the packet, \texttt{etherInputHook()} returns a non-zero value and the driver considers the packet to be delivered. If the \texttt{etherInputHook()} returns 0, the driver hands the packet to the TCP/IP stack.

On packet transmission, an installed \texttt{etherOutputHook()} receives a packet just before it would have been transmitted. If \texttt{etherOutputHook()} decides to prevent the packet from passing on, \texttt{etherOutputHook()} returns a non-zero value and the driver considers the packet to be transmitted. If the \texttt{etherOutputHook()} returns 0, the driver transmits the packet.

It is only possible to install one \texttt{etherInputHook()} and one \texttt{etherOutputHook()} routine per driver. This limits the number of alternate protocols to one, unless these \texttt{ether*Hook()} routines then act as a multiplexor for more protocols.
For more information on etherhooks, see the Tornado User’s Guide, 1.0.1: G.4 Network Interface Hook Routines.

NOTE: END drivers do not support etherhooks and are the only supported type of network driver for VxWorks 6.x.

5.3 Upgrading Your Driver to 4.4 BSD

To upgrade a driver from 4.3 BSD to 4.4 BSD you must change how the driver uses `ether_attach()`. This routine is almost always called from the driver’s own `xxattach()` routine and is responsible for placing the driver’s entry points, listed in Table 5-1, into the `ifnet` structure that the TCP/IP protocol to track drivers. Consider the call to `ether_attach()` shown below:

```c
ether_attach
  (IFNET *) & pDrvCtrl->idr,
  unit,
  "xx",
  (FUNCPTR) NULL,
  (FUNCPTR) xxIoctl,
  (FUNCPTR) xxOutput,
  (FUNCPTR) xxReset
);
```

As arguments, this routine expects an Interface Data Record (IDR), a unit number, and a quoted string that is the name of the device, in this case, “xx”. The next four arguments are the function pointers to relevant driver routines.

The first function pointer references this driver’s `init()` routine, which this driver does not need or have. The second function pointer references the driver’s `ioctl()` interface, which allows the upper layer to manipulate the device state. The third function pointer references the routine that outputs packets on the physical medium. The last function pointer references a routine that can reset the device if the TCP/IP stack decides that this needs to be done.

In 4.4 BSD, there is a generic output routine called `ether_output()` that all Ethernet device drivers can use. Thus, to convert the above `ether_attach()` call to a 4.4-style call, you would call `ether_attach()` as follows:
ether_attach
{
  (IFNET *) & pDrvCtrl->idr,
  unit,
  "xx",
  (FUNCPTR) NULL,
  (FUNCPTR) xxIoctl,
  (FUNCPTR) ether_output, /* generic ether_output */
  (FUNCPTR) xxReset
};
pDrvCtrl->idr.ac_if.if_start = (FUNCPTR)xxTxStartup;

This time, there is an extra line following the call to ether_attach(). This line of code adds a transmit startup routine to the Interface Data Record. The transmit startup routine is called by the TCP/IP stack after the generic ether_output() routine is called. This extra line of code assumes that the driver already has a transmit startup routine. If a driver lacks a separate transmit startup routine, you must write one. See the template in 5.3.4 Creating a Transmit Startup Routine.

5.3.1 Removing the xxOutput() Routine

If a 4.3 BSD driver has an xxOutput() routine, it should look something like the following:

static int xxOutput
{
  IDR * pIDR,
  MBUF * pMbuf,
  SOCK * pDestAddr
}
{
  return (ether_output ((IFNET *)pIDR,pMbuf, pDestAddr,
    (FUNCPTR) xxTxStartup, pIDR));
}

Internally, this routine calls the ether_output() routine, which expects a pointer to the startup routine as one of its arguments. However, in the 4.4 BSD model, all that work that is now handled in the TCP/IP stack. Thus, in a 4.4 BSD driver, this code is unnecessary and should be removed.

5.3.2 Changing the Transmit Startup Routine

Under 4.3 BSD, the routine prototype for a transmit startup routine is as follows:

static void xxTxStartup (int unit);
Under 4.4 BSD, the prototype has changed to the following:

```c
static void xxTxStartup (struct ifnet * pDrvCtrl);
```

The 4.4 BSD version expects a pointer to a driver control structure. This change eases the burden on the startup routine. Instead of having to find its own driver control structure, it receives a pointer to a driver control structure as input.

If the driver uses `netJobAdd()` to schedule the transmit startup routine for task-level execution, edit the `netJobAdd()` call to pass in a DRV_CTRL structure pointer instead of a unit number.

### 5.3.3 Adapting to Changes In Receiving Packets

Under 4.3 BSD, the driver calls `do_protocol_with_type()`. For example:

```c
do_protocol_with_type (etherType, pMbuf, &pDrvCtrl->idr, len);
```

This call expects an `etherType` (which the driver had to discover previously), a pointer to an `mbuf` containing the packet data, the Interface Data Record, and the length of the data.

Under 4.4 BSD, replace the call above with a call to `do_protocol()`. For example:

```c
do_protocol (pEh, pMbuf, &pDrvCtrl->idr, len);
```

The first parameter expects a pointer to the very beginning of the packet (including the link level header). All the other parameters remain the same. The driver no longer needs to determine the `etherType` for the protocol.

### 5.3.4 Creating a Transmit Startup Routine

Some 4.3 BSD drivers did not have a transmit startup routine. For such a driver, you must create one. The template is as follows:

```c
void templateStartup
(
   DRV_CTRL *pDrvCtrl
)
{
   MBUF * pMbuf;
   int length;
   TFD * pTfd;

   /*
   * Loop until there are no more packets ready to send or we
   * have insufficient resources left to send another one.
   */
   
   /*
   * ...
   */
}
```
while (pDrvCtrl->idr.ac_if.if_snd.ifq_head)
{
    /* Dequeue a packet from the send queue. */
    IF_DEQUEUE (&pDrvCtrl->idr.ac_if.if_snd, pMbuf);

    /*
    * Device specific code to get transmit resources, such as a
    * transmit descriptor, goes here.
    *
    */
    if (Insufficient Resources)
    {
        m_freem (pMbuf); /* Make sure to free the packet. */
        return;
    }

    /*
    * pData below is really the place in your descriptor,
    * transmit descriptor, or equivalent, where the data is
    * to be placed.
    */
    copy_from_mbufs (pData, pMbuf, length);

    if ((etherOutputHookRtn != NULL) &&
        (* etherOutputHookRtn)
        (&pDrvCtrl->idr, (ETH_HDR *)pTfd->enetHdr, length))
        continue;

    /*
    * Do hardware manipulation to set appropriate bits
    * and other stuff to get the packet to actually go.
    */

    /*
    * Update the counter that determines the number of
    * packets that have been output.
    */
    pDrvCtrl->idr.ac_if.if_opackets++;
}
} /* End of while loop. */
} /* End of transmit routine. */
5.4 Porting a BSD Ethernet Driver to the END Model

The MUX-based model for network drivers contains standardized entry points that are not present in the BSD model. Table 5-2 shows some of the analogies between routines found in BSD 4.3 drivers and those necessary for END drivers. Fortunately, you should be able to reuse much of the code from the BSD 4.3 network driver.

<table>
<thead>
<tr>
<th>END Entry Points</th>
<th>BSD 4.3 Style Entry Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>endLoad()</td>
<td>xxattach()</td>
</tr>
<tr>
<td>endUnload()</td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>xxReceive()</td>
</tr>
<tr>
<td>endSend()</td>
<td>xxOutput()</td>
</tr>
<tr>
<td>endIoctl()</td>
<td>xxIoctl()</td>
</tr>
<tr>
<td>endMCastAddrAdd()</td>
<td>N/A</td>
</tr>
<tr>
<td>endMCastAddrDel()</td>
<td>N/A</td>
</tr>
<tr>
<td>endMCastAddrGet()</td>
<td>N/A</td>
</tr>
<tr>
<td>endPollSend()</td>
<td>N/A</td>
</tr>
<tr>
<td>endPollReceive()</td>
<td>N/A</td>
</tr>
</tbody>
</table>

⚠️ CAUTION: When converting a BSD 4.3 network driver code to an END driver, you must replace all calls into the protocol with appropriate calls to the MUX. Also, you must remove all code that implements or uses the etherInputHook() and etherOutputHook() routines.

5.4.1 Rewriting xxattach() to Use an endLoad() Interface

Rewrite the interface of your xxattach() to match the endLoad() entry point described in Loading the Device: endLoad(), p. 104.
Much of the code that handles the specifics of hardware initialization should be the same. However, when allocating the memory for packet reception buffers that are passed up to the protocol, you should use the MUX buffer management utilities. See *Setting Up and Using Memory for Receive and Transmit Buffers*, p.64, as well as the reference entry for `muxBufInit()`. Remove any code your `xxattach()` included to support the implementation of the `etherInputHook()` and `etherOutputHook()` routines.

### 5.4.2 Handling Task-Level Packet Reception with `xxReceive()`

Because the MUX does not directly call the driver’s packet reception code, there is no `endReceive()` entry point. However, your driver still needs to handle packet reception at the task level. Unfortunately, most of the code in this driver routine requires extensive revision. Instead of calling the protocol directly, this routine uses a MUX-supplied routine to pass a packet up to the protocol (see *Handling Packet Reception*, p.69). Also, your receive routine should use the MUX-managed memory pool as its receive buffer area.

### 5.4.3 Rewriting `xxOutput()` to Use an `endSend()` Interface

Rewrite the interface of your `xxOutput()` to match the `endSend()` routine described in *Sending Data Out on the Device: endSend()*, p.110.

Much of the code that dealt directly with putting the packet on the hardware should need little if any revision. However, you should change your code to use `mblk` chains allocated out of an `endBufLib`-managed memory pool. For more information, see the reference entry for `netBufLib`.

### 5.4.4 Using `xxioctl()` as a Basis for `endioctl()`

Rewrite the interface of your `xxioctl()` to match the `endioctl()` routine described in *Providing an Opaque Control Interface to Your Driver: endioctl()*, p.108. If your driver used `xxioctl()` to implement multicasting, you must separate those operations into the separate `endMCastAddrAdd()`, `endMCastAddrDel()`, and `endMCastAddrGet()` routines.
6.1 Introduction

TrueFFS is an optional product that allows a file system to be used and maintained on flash media. TrueFFS provides a number of features that enhance the performance of the flash media that is used to contain the file system, and also allow the same flash bank to contain bootable images or other constant data. For more information on TrueFFS features, see the VxWorks Hardware Considerations Guide. For details on configuring and using TrueFFS with a BSP that includes TrueFFS support, see the VxWorks Kernel Programmer’s Guide.

This chapter contains information necessary to write routines for TrueFFS support of new devices.
6.2 TrueFFS Overview

This section provides a brief overview of the TrueFFS layers. The individual layers are discussed in greater detail in later sections. For a graphical presentation of a flash device layout, see Figure 6-9.

TrueFFS is composed of a core layer and three functional layers—the translation layer, the memory technology driver (MTD) layer, and the socket layer—as illustrated in Figure 6-1. The three functional layers are provided in source code form, in binary form, or in both, as noted in the following sections.

6.2.1 Core Layer

The core layer connects other layers to each other. In addition, this layer channels work to the other layers and handles global issues, such as backgrounding, garbage collection, timers, and other system resources. The core layer is provided in binary form only.

6.2.2 MTD Layer

The memory technology driver (MTD) implements the low-level programming of the flash medium. This includes map, read, write, and erase functionality. MTDs are provided in both source and binary form.
6.2.3 Socket Layer

The socket layer provides the interface between TrueFFS and the board hardware, providing board-specific hardware access routines. This layer is responsible for power management, card detection, window management, and socket registration. TrueFFS socket drivers are provided in source code only.

6.2.4 Flash Translation Layer

The flash translation layer (FTL) maintains the map that associates the file system’s view of the storage medium with the erase blocks in flash. The block allocation map (BAM) is the basic building block for implementing wear-leveling and error recovery. The translation layer is media specific (NOR or SSFDC) and is provided in binary form only.

6.3 The TrueFFS Driver Development Process

This section provides detailed information on the MTD, socket, and flash translation layers of TrueFFS. This information is intended to aid you in the TrueFFS driver development process. Detailed TrueFFS usage information is available in the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS.

6.3.1 Using MTD-Supported Flash Devices

Standard MTDs are written to support multiple device types and multiple configurations, without change to the source code. This feature comes with a cost to performance. If you choose to customize your MTD to a specific flash device and configuration, you can greatly increase performance when compared to the generic MTDs provided with this product.

NOTE: File systems are typically slow. In most cases, the performance increase that can be obtained by optimizing the MTD does not merit the effort to produce and support the optimized version.
When customization of TrueFFS is required, the most common modification is to provide a custom MTD. This usually occurs because the standard product does not support the flash parts chosen for the project, but it may also be because enhanced performance is required. If you are customizing an existing, working MTD, you can use the standard version as a reference and remove extraneous material as necessary.

The following sections list the flash devices that are supported by the MTDs provided with TrueFFS.

**Supporting the Common Flash Interface (CFI)**

TrueFFS supports devices that conform to the *common flash interface* (CFI) specification. This includes the following command sets:

- **Intel/Sharp CFI Command Set**: This is the CFI specification listing for the *scalable command set* (CFI/SCS). The driver file for this MTD is `installDir/vxworks-6.x/target/src/drv/tffs/cfiscs.c`. Support for this command set is largely derived from *Application Note 646*, available at the Intel Web site.

- **AMD/Fujitsu CFI Command Set**: This is the *Embedded Program Algorithm* and flexible sector architecture listing for the SCS command set. The driver file for this MTD is `installDir/vxworks-6.x/target/src/drv/tffs/cfiamd.c`. Support details for this MTD are described in *AMD/Fujitsu CFI Flash Support*, p.146.

Devices that require support for both command sets are rare. Therefore, to facilitate code readability, Wind River provides support for each command set in a separate MTD. To support both command sets, you must configure your system to include both MTDs. (For more information, see the *VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS*).

**Common Functionality**

Both MTDs support 8- and 16-bit devices, and 8- and 16-bit wide interleaves. Configuration macros (which are described in the code) are used to control configuration settings, and must be defined specifically for your system. If you modify the MTD code, it must be rebuilt. In particular, you may need to address the following macros:

**INTERLEAVED_MODEQUIRES_32BIT_WRITES**

Must be defined for systems that have 16-bit interleaves and require support for the “write-to-buffer” command.
SAVE_NVRAM_REGION
Excludes the last erase block on each flash device in the system that is used by TrueFFS; this is so that the region can be used for Non-Volatile Storage of boot parameters.

CFI_DEBUG
Makes the driver verbose by using the I/O routine defined by DEBUG_PRINT.

BUFFER_WRITE_BROKEN
Introduced to support systems that registered a buffer size greater than 1, yet could not support writing more than a byte or word at a time. When defined, it forces the buffer size to 1.

DEBUG_PRINT
If defined, makes the driver verbose by using its value.

NOTE: These macros can only be configured by defining them in the MTD source file, they cannot be configured using the project facility.

CFI/SCS Flash Support
The MTD defined in cfiscs.c supports flash components that follow the CFI/SCS specification. CFI is a standard method for querying flash components for their characteristics. SCS is a second layer built on the CFI specification. This lets a single MTD handle all CFI/SCS flash technology in a common manner.

NOTE: The cfiscs.c file is provided as an example only. Any current BSP that uses an MTD for one of these chips provides a custom MTD in the BSP directory.

The joint CFI/SCS specification is currently used by Intel Corporation and Sharp Corporation for all new flash components (starting in 1997).

The CFI document can be downloaded from:
http://www.intel.com/design/flcomp/applnots/292204.htm
or can be found by searching for CFI at:
http://www.intel.com/design

You must define the INCLUDE_MTD_CFISCS macro in your BSP sysTfs.c file to include this MTD in TrueFFS.

On some more recent target boards, non-volatile RAM circuitry does not exist and BSP developers have opted to use the high end of flash for this purpose. In this case, the last erase block of each flash part is used to make up this region. The
CFI/SCS MTD supports this concept by providing the compiler constant SAVE_NVRAM_REGION. If this constant is defined, the driver reduces the device’s size by a value equal to the erase block size times the number of devices; this results in an NVRAM region that is preserved and never over-written. ARM BSPs, in particular, use flash for NVRAM and for the boot image.

**AMD/Fujitsu CFI Flash Support**

In AMD and Fujitsu devices, the flexible sector architecture, also called boot block, is only supported when erasing blocks. However, because the MTD presents this division transparently, the TrueFFS core and translation layers have no knowledge of the subdivision. According to the data sheet for a 29LV160 device, the device is comprised of 35 sectors. However, the four boot block sectors appear to the core and translation layer as yet another, single (64 KB) sector. Thus, the TrueFFS core detects only 32 sectors. Consequently, the code that supports boot images also has no knowledge of the boot block, and cannot provide direct support for it.

AMD and Fujitsu devices also include a concept of top and bottom boot devices. However, the CFI interrogation process does not provide a facility for distinguishing between these two boot device types. Thus, in order to determine the boot block type, the driver for these devices embeds a Joint Electronic Device Engineering Council (JEDEC) device ID. This limits the number of supported devices to those that are registered in the driver and requires verification that the device in use is listed in the registry.

**Supporting Other MTDs**

If you are not using a CFI-compliant MTD, Wind River also provides the following MTDs.

**Intel 28F016 Flash Support**

The MTD defined in i28f016.c supports Intel 28F016SA and Intel 28F008SV flash components. Any flash array or card based on these chips is recognized and supported by this MTD. This MTD also supports interleaving factors of 2 and 4 for BYTE-mode 28F016 component access.

For WORD-mode component access, only non-interleaved (interleave 1) mode is supported. The list of supported flash media includes the following:

- Intel Series-2+ PC Cards
- M-Systems Series-2+ PC Cards
Define `INCLUDE_MTD_I28F016` in your BSP `sysTffs.c` file to include this MTD in TrueFFS.

### Intel 28F008 Flash Support

The MTD defined in `I28F008.c` supports the Intel 28F008SA, Intel 28F008SC, and Intel 28F016SA/SV (in 8 Mb compatibility mode) flash components. Any flash array or card based on these chips is recognized and supported by this MTD. However, the WORD-mode of 28F016SA/SV is not supported (BYTE-mode only). This MTD also supports all interleaving factors (1, 2, 4, ...). Interleaving of more than 4 is recognized, although the MTD does not access more than 4 flash parts simultaneously. The list of supported flash media includes the following:

- M-Systems D-Series PC Cards
- M-Systems S-Series PC Cards
- Intel Series-2 (8-mbit family) PC Cards
- Intel Series-2+ (16-mbit family) PC Cards
- Intel Value Series 100 PC Cards
- Intel Miniature cards
- M-Systems PC-FD, PC-104-FD, Tiny-FD flash disks

Define `INCLUDE_MTD_I28F008` in your BSP `sysTffs.c` file to include this MTD in TrueFFS.

### AMD/Fujitsu Flash Support

The MTD defined in `amdmtd.c` (8-bit) supports AMD flash components of the AMD Series-C and Series-D flash technology family, as well as the equivalent Fujitsu flash components. The flash types supported are:

- Am29F040 (JEDEC IDs 01a4h, 04a4h)
- Am29F080 (JEDEC IDs 01d5h, 04d5h)
- Am29LV080 (JEDEC IDs 0138h, 0438h)
- Am29LV008 (JEDEC IDs 0137h, 0437h)
- Am29F016 (JEDEC IDs 01adh, 04adh)
- Am29F016C (JEDEC IDs 013dh, 043dh)

Any flash array or card based on these chips is recognized and supported by this MTD. The MTD supports interleaving factors of 1, 2, and 4. The list of supported flash media includes the following:

- AMD and Fujitsu Series-C PC cards
- AMD and Fujitsu Series-D PC cards
- AMD and Fujitsu miniature cards
Define `INCLUDE_MTD_AMD` in your BSP `sysTffs.c` file to include the 8-bit MTD in TrueFFS.

**Obtaining Disk-On-Chip Support**

The previous demand for NAND devices has been in one of two forms: SSFDC/Smart Media devices and Disk On Chip from M-Systems. Each of these forms is supported by a separate translation layer. Support for M-Systems devices must now be obtained directly from M-Systems and is no longer distributed with the VxWorks product. This allows M-Systems to add Disk On Chip specific optimizations within TrueFFS without affecting other supported devices. Current versions of VxWorks only support NAND devices that conform to the SSFDC specification (for more information, see the *VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS*).

**6.3.2 Writing MTD Components**

An MTD is a software module that provides TrueFFS with data, and with pointers to the routines that it uses to program the flash memory. All MTDs must provide the following three routines: a write routine, an erase routine, and an identification routine. The MTD module uses an identification routine to evaluate whether the type of the flash device is appropriate for the MTD. If you are writing your own MTD, you need to define it as a component and register the identification routine.

For source code examples of MTDs, see the `installDir/vxworks-6.x/target/src/drv/tffs` directory.

**Writing the MTD Identification Routine**

TrueFFS provides a flash structure in which information about each flash part is maintained. The identification process is responsible for setting up the flash structure correctly.

**NOTE:** Many of the MTDs previously developed by M-Systems or Wind River are provided in source form as examples of how you should write an MTD (in `installDir/vxworks-6.x/target/src/drv/tffs`). This section provides additional information about writing identification routines.
In the process of creating a logical block device for a flash memory array, TrueFFS tries to match an MTD to the flash device. To do this, TrueFFS calls the identification routine from each MTD until one reports a match. The first reported match is the one taken. If no MTD reports a match, TrueFFS falls back on a default read-only MTD that reads from the flash device by copying from the socket window.

The MTD identification routine is guaranteed to be called prior to any other routine in the MTD. An MTD identification routine is of the following format:

```c
FLStatus xxxIdentify(FLFlash vol)
```

Within an MTD identify routine, you must probe the device to determine its type. How you do this depends on the hardware. If the type is not appropriate to this MTD, return a failure. Otherwise, set the members of the `FLFlash` structure listed below (see `Initializing the FLFlash Structure Members`, p.149).

The identification routine for every MTD must be registered in `mtdTable[ ]` defined in `installDir/vxworks-6.x/target/src/drv/tffs/tffsConfig.c`. Each time a volume is mounted, the list of identification routines is traversed to find the MTD suitable for the volume. This provides better service for hot-swap devices; no assumption is made about a previously identified device being the only device that works for a given volume.

Device identification can be done in a variety of ways. If your device conforms to JEDEC or CFI standards, you can use the identification process provided for the device. You may want your MTD to identify many versions of the device, or only one.

### Initializing the FLFlash Structure Members

At the end of the identification process, the ID routine needs to set all data elements in the `FLFlash` structure, except the `socket` member. The `socket` member is set by routines internal to TrueFFS. The `FLFlash` structure is defined in `installDir/vxworks-6.x/target/h/tffs/flflash.h`. Members of this structure are the following:

- **type**
  - The JEDEC ID for the flash memory hardware. This member is set by the MTD identification routine.

- **erasableBlockSize**
  - The size, in bytes, of an erase block for the attached flash memory hardware. This value takes interleaving into account. Thus, when setting this value in an MTD, the code is often of the following form:

    ```c
    vol.erasableBlockSize = aValue * vol.interleaving;
    ```
Where \textit{aValue} is the erasable block size of a flash chip that is not interleaved with another.

\textbf{chipSize}

The size (storage capacity), in bytes, of one of the flash memory chips used to construct the flash memory array. This value is set by the MTD, using your \texttt{flFitInSocketWindow()} global routine.

\textbf{noOfChips}

The number of flash memory chips used to construct the flash memory array.

\textbf{interleaving}

The interleaving factor of the flash memory array. This is the number of devices that span the data bus. For example, on a 32-bit bus we can have four 8-bit devices or two 16-bit devices.

\textbf{flags}

Bits 0-7 are reserved for TrueFFS use (TrueFFS uses these flags to track items such as the volume mount state). Bits 8-15 are reserved for MTD use.

\textbf{mtdVars}

This field, if used by the MTD, is initialized by the MTD identification routine to point to a private storage area. These are instance-specific. For example, suppose you have an Intel RFA based on the I28F016 flash part and you also have a PCMCIA socket into which you decide to plug a card that has the same flash part. The same MTD is used for both devices, and the \texttt{mtdVars} are used for the variables that are instance-specific, so that an MTD may be used more than once in a system.

\textbf{socket}

This member is a pointer to the \texttt{FLSocket} structure for your hardware device. This structure contains data and pointers to the socket layer routines that TrueFFS needs to manage the board interface for the flash memory hardware. The routines referenced in this structure are installed when you register your socket driver (see \texttt{6.3.3 Socket Drivers}, p.157). Further, because TrueFFS uses these socket driver routines to access the flash memory hardware, you must register your socket driver \textit{before} you try to run the MTD identify routine that initializes the bulk of this structure.

\textbf{map}

A pointer to the flash memory map routine, the routine that maps flash into an area of memory. Internally, TrueFFS initializes this member to point to a default map routine appropriate for all NOR (linear) flash memory types. This default routine maps flash memory through simple socket mapping. Flash
should replace this pointer to the default routine with a reference to a routine that uses map-through-copy emulation.

**read**

A pointer to the flash memory read routine. On entry to the MTD identification routine, this member has already been initialized to point to a default read routine that is appropriate for all NOR (linear) flash memory types. This routine reads from flash memory by copying from a mapped window. If this is appropriate for your flash device, leave **read** unchanged. Otherwise, your MTD identify routine must update this member to point to a more appropriate routine.

**write**

A pointer to the flash memory write routine. Because of the dangers associated with an inappropriate write routine, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member.

**erase**

A pointer to the flash memory erase routine. Because of the dangers associated with an inappropriate erase routine, the default routine for this member returns a write-protect error. The MTD identification routine must supply an appropriate function pointer for this member.

**setPowerOnCallback**

A pointer to the routine TrueFFS should execute after the flash hardware device powers up. TrueFFS calls this routine when it tries to mount a flash device. Do not confuse this member of **FLFlash** with the **powerOnCallback** member of the **FLSocket** structure. For many flash memory devices, no such routine is necessary.

**Return Value**

The identification routine must return **flOK** or an appropriate error code defined in **flbase.h**. The stub provided is:

```c
FLStatus myMTDIdentification
   (FLFlash vol)
{
  /* Do what is needed for identification */
  /* If identification fails return appropriate error */
  return flOK;
}
```
After setting the members listed above, this routine should return \texttt{flOK}.

**Call Sequence**

Upon success, the identification routine updates the \texttt{FLFlash} structure, which also completes the initialization of the \texttt{FLSocket} structure referenced within this \texttt{FLFlash} structure.

**Writing the MTD Map Routine**

MTDs need to provide a map routine only when a RAM buffer is required for windowing. No MTDs are provided for devices of this kind in this release. If the device you are using requires such support, you need to add a map routine to your MTD and assign a pointer to it in \texttt{FLFlash.map}. The routine takes three arguments, a pointer to the volume structure, a “card address”, and a length field, and returns a void pointer.
static void FAR0 * Map
    (FLFlash vol,
     CardAddress address,
     int length
    )
{
    /* implement function */
}

### Writing the MTD Read, Write, and Erase Routines

Typically, your read, write, and erase routines should be as generic as possible. This means that they should:

- Read, write, or erase only a byte, a word, or a long word at a time.
- Be able to handle an unaligned read or write.
- Be able to handle a read, write, or erase that crosses chip boundaries.

When writing these routines, you probably want to use the MTD helper routines `flNeedVpp()`, `flDontNeedVpp()`, and `flWriteProtected()`. The interfaces for these routines are as follows:

```
FLStatus  flNeedVpp(FLSocket vol)
void      flDontNeedVpp(FLSocket vol)
FLBoolean flWriteProtected(FLSocket vol)
```

Use `flNeedVpp()` if you need to turn on the Vpp (the programming voltage) for the chip. Internally, `flNeedVpp()` bumps a counter, `FLSocket.VppUsers`, and then calls the routine referenced in `FLSocket.VppOn`. After calling `flNeedVpp()`, check its return status to verify that it succeeded in turning on Vpp.

When done with the write or erase that required Vpp, call `flDontNeedVpp()` to decrement the `FLSocket.VppUsers` counter. This `FLSocket.VppUsers` counter is part of a delayed-off system. While the chip is busy, TrueFFS keeps the chip continuously powered. When the chip is idle, TrueFFS turns off the voltage to conserve power. ¹

Use `flWriteProtected()` to test that the flash device is not write protected. The MTD write and erase routines must not do any flash programming before checking that writing to the card is allowed. The boolean routine `flWriteProtected()` returns TRUE if the card is write-protected and FALSE otherwise.

¹ An MTD does not need to touch Vcc. TrueFFS turns Vcc on before calling an MTD routine.
Read Routine

If the flash device can be mapped directly into flash memory, it is generally a simple matter to read from it. TrueFFS supplies a default routine that performs a remap, and simple memory copy, to retrieve the data from the specified area. However, if the mapping is done through a buffer, you must provide your own read routine.

Write Routine

The write routine must write a given block at a specified address in flash. Its arguments are a pointer to the flash device, the address in flash to write to, a pointer to the buffer that must be written, and the buffer length. The last parameter is boolean, and if set to TRUE implies that the destination has not been erased prior to the write request. The routine is declared as static since it is only called from the volume descriptor. The stub provided is:

```c
static FLStatus myMTDWrite
(  
  FLFlash vol,
  CardAddress address,
  const void FAR1 *buffer,
  int length,
  FLBoolean overwrite
)
{
  /* Write routine */
  return flOK;
}
```

The write routine must do the following:

- Check to see if the device is write protected.
- Turn on Vpp by calling `flNeedVpp()`.
- Always “map” the “card address” provided to a `flashPtr` before you write.

When implementing the write routine, iterate through the buffer in a way that is appropriate for your environment. If writes are permitted only on word or double word boundaries, check to see whether the buffer address and the card address are so aligned. Return an error if they are not.

The correct algorithms usually follow a sequence in which you:

- Issue a “write setup” command at the card address.
- Copy the data to that address.
- Loop on the status register until either the status turns `OK` or you time out.
Device data sheets usually provide flow charts for this type of algorithm. AMD
devices require an unlock sequence to be performed as well.

The write routine is responsible for verifying that what was written matches the
content of the buffer from which you are writing. The file flsystem.h has
prototypes of compare routines that can be used for this purpose.

Erase Routine

The erase routine must erase one or more contiguous blocks of a specified size.
This routine is given a flash volume pointer, the block number of the first erasable
block and the number of erasable blocks. The stub provided is:

```c
Static FLStatus myMTErase
(
    FLFlash vol,
    int firstBlock,
    int numOfBlocks
) {
    volatile UINT32 * flashPtr;
    int iBlock;

    if (flWriteProtected(vol.socket))
        return flWriteProtected;
    for (iBlock = firstBlock; iBlock < iBlock + numOfBlocks; Iblock++)
        {  
            flashPtr = vol.map (&vol, iBlock * vol.erasableBlockSize, 0);
            /* Perform erase operation here */
            /* Verify if erase succeeded */
            /* return flWriteFault if failed*/
        }
    return flOK;
}
```

As input, the erase can expect a block number. Use the value of the
erasableBlockSize member of the FLFlash structure to translate this block number
to the offset within the flash array.

Defining Your MTD as a Component

Once you have completed the MTD, you need to add it as a component to your
system project. By convention, MTD components are named
INCLUDE_MTD_someName; for example, INCLUDE_MTD_USR. You can include
the MTD component either through the project facility or, for a command-line
configuration and build, by defining it in the socket driver file, sysTffs.c.
Adding Your MTD to the Project Facility

In order to have the MTD recognized by the project facility, a component description of the MTD is required. To add your own MTD component to your system by using the project facility, edit the file `installDir/vxworks-6.x/target/config/comps/vxworks/00tffs.cdf` to include it. MTD components are defined in that file using the following format:

```c
Component INCLUDE_MTD_type
{
    NAME
    SYNOPSIS type devices
    MODULES filename.o
    HDR_FILES tffs/flflash.h tffs/backdrnd.h
    REQUIRES INCLUDE_TFFS \n        INCLUDE_TL_type
}
```

Once you define your MTD component in the `00tffs.cdf` file, it appears in the project facility the next time you run Workbench.

Defining the MTD in the Socket Driver File

For a command-line configuration and build, you can include the MTD component simply by defining it in the socket driver file, `sysTffs.c`, as follows:

```
#define INCLUDE_MTD_USR
```

Add your MTD definition to the list of those defined between the conditional clause, as described in the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS. Then, define the correct translation layer for your MTD. If both translation layers are defined in the socket driver file, undefine the one you are not using. If both are undefined, define the correct one. For other examples, see the `type=sysTffs.c` files in `installDir/vxworks-6.x/target/src/drv/tffs/sockets`.

⚠️ **CAUTION:** Be sure that you have the correct `sysTffs.c` file before changing the defines. For more information, see Porting the Socket Driver Stub File, p.158.

Registering the Identification Routine

The identification routine for every MTD must be registered in `mtdTable[]`. Each time a volume is mounted, TrueFFS searches this list to find an MTD suitable for the volume (flash device). For each component that has been defined for your system, TrueFFS executes the identification routine referenced in `mtdTable[]`, until
it finds a match to the flash device. The current `mtdTable[]` as defined in `installDir/vxworks-6.x/target/src/drv/tffs/tffsConfig.c` is:

```c
MTDidentifyRoutine mtdTable[] = /* MTD tables */
{
    #ifdef INCLUDE_MTD_I28F016
        i28f016Identify,
    #endif /* INCLUDE_MTD_I28F016 */

    #ifdef INCLUDE_MTD_I28F008
        i28f008Identify,
    #endif /* INCLUDE_MTD_I28F008 */

    #ifdef INCLUDE_MTD_AMD
        amdMTDIdentify,
    #endif /* INCLUDE_MTD_AMD */

    #ifdef INCLUDE_MTD_CDSN
        cdsnIdentify,
    #endif /* INCLUDE_MTD_CDSN */

    #ifdef INCLUDE_MTD_DOC2
        doc2Identify,
    #endif /* INCLUDE_MTD_DOC2 */

    #ifdef INCLUDE_MTD_CFISCS
        cfiscsIdentify,
    #endif /* INCLUDE_MTD_CFISCS */
};
```

If you write a new MTD, list its identification routine in `mtdTable[]`. For example:

```c
#ifdef INCLUDE_MTD_USR
    usrMTDIdentify,
#endif /* INCLUDE_MTD_USR */
```

It is recommended that you surround the component name with conditional include statements, as shown above. The symbolic constants that control these conditional includes are defined in the BSP `config.h` file. Using these constants, your end users can conditionally include specific MTDs.

When you add your MTD identification routine to this table, you should also add a new constant to the BSP `config.h` file.

### 6.3.3 Socket Drivers

The socket driver is implemented in the file `sysTffs.c`. TrueFFS provides a stub version of the socket driver file for BSPs that do not include one. As a writer of the socket driver, your primary focus is on the following key contents of the socket driver file:
The `sysTffsInit()` routine, the main routine. This routine calls the socket registration routine.

The `xxxRegister()` routine, the socket registration routine. This routine is responsible for assigning routines to the member functions of the socket structure.

The routines assigned by the registration routine.

The macro values that should reflect your hardware.

In this stub file, all of the required routines are declared. Most of these routines are defined completely, although some use generic or fictional macro values that you may need to modify.

The socket register routine in the stub file is written for RFA (resident flash array) sockets only. There is no stub version of the registration routine for PCMCIA socket drivers. If you are writing a socket driver for RFA, you can use this stub file and follow the steps described in the following section. If you are writing a PCMCIA socket driver, see the example in `installDir/vxworks-6.x/target/src/drv/tffs/sockets/pc386-sysTffs.c` and the general information in Understanding Socket Driver Functionality, p.162.

**NOTE:** Examples of other RFA socket drivers are in `installDir/vxworks-6.x/target/src/drv/tffs/sockets`.

**Porting the Socket Driver Stub File**

If you are writing your own socket driver, it is assumed that your BSP does not provide one. When you run the build, a stub version of the socket driver, `sysTffs.c`, is copied from `installDir/vxworks-6.x/target/config/comps/src` to your BSP directory. Alternatively, you can copy this version manually to your BSP directory before you run a build. In either case, edit only the file copied to the BSP directory; do not modify the original stub file.

This stub version is the starting point to help you port the socket driver to your BSP. As such, it contains incomplete code and does not compile. The modifications you need to make are listed below. The modifications are not extensive and all are noted by /* TODO */ clauses.

1. Replace “fictional” macro values, such as `FLASH_BASE_ADRS`, with correct values that reflect your hardware. Then, remove the following line:

   ```c
   #error "sysTffs: Verify system macros and function before first use"
   ```
2. Add calls to the registration routine for each additional device that your BSP supports. Therefore, if you have only one device, you do not need to do anything for this step. For details, see the following section.

3. Review the implementation for the two routines marked /* TODO */. You may or may not need to add code for them. For details, see Implementing the Socket Structure Member Functions, p. 159.

**CAUTION:** Do not edit the original copy of the stub version of sysTffs.c in installDir/vxworks-6.x/target/config/comps/src. You may need it for future ports.

**Calling the Socket Register Routines**

The main routine in sysTffs.c is sysTffsInit(), which is automatically called at boot time. The last lines of this routine call the socket register routines for each device supported by your system. The stub sysTffs.c file specifically calls the socket register routine rfaRegister().

If your BSP supports only one (RFA) flash device, you do not need to edit this section. However, if your BSP supports several flash devices, you must edit the stub file to add calls for each socket’s register routine. The place to do this is indicated by the /* TODO */ comments in the sysTffsInit() routine.

If you have several socket drivers, you can encapsulate each xxxRegister() call in pre-processor conditional statements, as in the following example:

```c
#ifdef INCLUDE_SOCKET_PCIC0
  (void) pcRegister (0, PC_BASE_ADRS_0); /* flash card on socket 0 */
#endif /* INCLUDE_SOCKET_PCIC0 */

#ifdef INCLUDE_SOCKET_PCIC1
  (void) pcRegister (1, PC_BASE_ADRS_1); /* flash card on socket 1 */
#endif /* INCLUDE_SOCKET_PCIC1 */
```

Define the constants in the BSP sysTffs.c. Then, you can use them to selectively control which calls are included in sysTffsInit() at compile time.

**Implementing the Socket Structure Member Functions**

The stub socket driver file also contains the implementation for the rfaRegister() routine. This routine assigns routines to the member functions of the FLSocket structure, vol. TrueFFS uses this structure to store the data and function pointers that handle the hardware (socket) interface to the flash device. For the most part, you need not be concerned with the FLSocket structure, only with the routines assigned to it. Once these routines are implemented, you never call them directly. They are called automatically by TrueFFS.
All of the routines assigned to the socket structure member functions by the registration routine are defined in the stub socket driver module. However, only the `rfaSocketInit()` and `rfaSetWindow()` routines are incomplete. When you are editing the stub file, note the `#error` and `/* TODO */` comments in the code. These indicate where and how you need to modify the code.

Following is a list of all of the routines assigned by the registration routine, along with a description of how each is implemented in the stub file. The two routines that require your attention are listed with descriptions of how they should be implemented.

**rfaCardDetected()**

This routine always returns TRUE in RFA environments because the device is not removable. Implementation is complete in the stub file.

**rfaVccOn()**

Vcc must be known to be good on exit. It is assumed to be ON constantly in RFA environments. This routine is simply a wrapper. While the implementation is complete in the stub file, you may want to add code as described below.

When switching Vcc on, the `rfaVccOn()` routine must not return until Vcc has stabilized at the proper operating voltage. If necessary, your routine should delay execution with an idle loop, or with a call to the `flDelayMsec()` routine, until the Vcc has stabilized.

**rfaVccOff()**

Vcc is assumed to be ON constantly in RFA environments. This routine is simply a wrapper and is complete in the stub file.

**rfaVppOn()**

Vpp must be known to be good on exit and is assumed to be ON constantly in RFA environments. This routine is not optional, and must always be implemented. Do not delete this routine. While the implementation in the stub file is complete, you may want to add code, as described below.

When switching Vpp on, the `rfaVppOn()` routine must not return until Vpp has stabilized at the proper voltage. If necessary, your `VppOn()` routine should delay execution with an idle loop or with a call to the `flDelayMsec()` routine, until the Vpp has stabilized.

**NOTE:** More detailed information on the functionality of each routine is provided in *Understanding Socket Driver Functionality*, p.162. However, this information is not necessary for you to port the socket driver.
6 Flash File System Support with TrueFFS
6.3 The TrueFFS Driver Development Process

rfaVppOff()
Vpp is assumed to be ON constantly in RFA environments. This routine is complete in the stub file; however, it is not optional, and must always be implemented. Therefore, do not delete this routine.

rfaSocketInit()
Contains a /* TODO */ clause.

This routine is called each time TrueFFS is initialized (the drive is accessed). It is responsible for ensuring that the flash is in a usable state (that is, board-level initialization). If, for any reason, there is something that must be done prior to such an access, this is the routine in which you perform that action. For more information, see rfaSocketInit() in Socket Member Functions, p.163.

rfaSetWindow()
Contains a /* TODO */ clause.

This routine uses the FLASH_BASE_ADRS and FLASH_SIZE values that you set in the stub file. As long as those values are correct, the implementation for this routine in the stub file is complete.

TrueFFS calls this routine to initialize key members of the window structure, which is a member of the FLSocket structure. For most hardware, the setWindow() routine does the following, which is already implemented in the stub file:

Sets the window.baseAddress to the base address in terms of 4 KB pages.

Calls flSetWindowSize(), specifying the window size in 4 KB units (window.baseAddress). Internally, the call to flSetWindowSize() sets window.size, window.base, and window.currentPage for you.

This routine sets current window hardware attributes: base address, size, speed and bus width. The requested settings are given in the vol.window structure. If it is not possible to set the window size requested in vol.window.size, the window size should be set to a larger value, if possible. In any case, vol.window.size should contain the actual window size (in 4 KB units) on exit.

For more information, see rfaSetWindow() in Socket Member Functions, p.163 and Socket Windowing and Address Mapping, p.165.

⚠️ CAUTION: On systems with multiple socket drivers (to handle multiple flash devices), make sure that the window base address is different for each socket. In addition, the window size must be taken into account to verify that the windows do not overlap.
rfaSetMappingContext()

TrueFFS calls this routine to set the window mapping register. Because board-resident flash arrays usually map the entire flash in memory, they do not need this routine. In the stub file it is a wrapper, thus implementation is complete.

rfaGetAndClearChangeIndicator()

This routine always returns FALSE in RFA environments because the device is not removable. This routine is complete in the stub file.

rfaWriteProtected()

This routine always returns FALSE for RFA environments. It is completely implemented in the stub file.

Understanding Socket Driver Functionality

Socket drivers in TrueFFS are modeled after the PCMCIA socket services. They must provide the following:

- services that control power to the socket (be it PCMCIA, RFA, or any other type)
- criteria for setting up the memory windowing environment
- support for card change detection
- a socket initialization routine

This section describes details about socket registration, socket member functions, and the windowing and address mapping set by those routines. This information is not necessary to port the stub RFA file; however, it may be useful for writers of PCMCIA socket drivers.

Socket Registration

The first task the registration routine performs is to assign drive numbers to the socket structures. This is fully implemented in the stub file. You only need to be aware of the drive number when formatting the drives (for more information, see the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS).

The drive numbers are index numbers into a pre-allocated array of FLSocket structures. The registration sequence dictates the drive number associated with a drive, as indicated in the first line of code from the rfaRegister() routine:

```c
FLSocket vol = f1Socket0f (no0fDrives);
```
Here, $\text{noOfDrives}$ is the running count of drives attached to the system. The routine $\text{flSocketOf}()$ returns a pointer to socket structure, which is used as the volume description and is incremented by each socket registration routine called by the system. Thus, the TrueFFS core in the socket structures are allocated for each of the (up to) 5 drives supported for the system.\footnote{TrueFFS only supports a maximum of 5 drives numbered 0-4.} When TrueFFS invokes the routines that you implement to handle its hardware interface needs, it uses the drive number as an index into the array to access the socket hardware for a particular flash device.

**Socket Member Functions**

**rfaCardDetected()**

This routine reports whether there is a flash memory card in the PCMCIA slot associated with this device. For non-removable media, this routine should always return TRUE. Internally, TrueFFS calls this routine every 100 milliseconds to check that flash media is still there. If this routine returns FALSE, TrueFFS sets $\text{cardChanged}$ to TRUE.

**rfaVccOn()**

TrueFFS can call this routine to turn on $\text{Vcc}$, which is the operating voltage. For the flash memory hardware, $\text{Vcc}$ is usually either 5 or 3.3 Volts. When the media is idle, TrueFFS conserves power by turning $\text{Vcc}$ off at the completion of an operation. Prior to making a call that accesses flash memory, TrueFFS uses this routine to turn the power back on.

When socket polling is active, a delayed $\text{Vcc}$-off mechanism is used, in which $\text{Vcc}$ is turned off only after at least one interval has passed. If several flash-accessing operations are executed in rapid sequence, $\text{Vcc}$ remains on during the sequence, and is turned off only when TrueFFS goes into a relatively idle state.

**rfaVccOff()**

TrueFFS can call this routine to turn off the operating voltage for the flash memory hardware. When the media is idle, TrueFFS conserves power by turning $\text{Vcc}$ off. However, when socket polling is active, $\text{Vcc}$ is turned off only after a delay. Thus, if several flags accessing operations are executed in rapid sequence, $\text{Vcc}$ is left on during the sequence. $\text{Vcc}$ is turned off only when TrueFFS goes into an idle state. $\text{Vcc}$ is assumed to be ON constantly in RFA environments.
**rfaVppOn()**

This routine is not optional, and must always be implemented. TrueFFS calls this routine to apply Vpp, which is the programming voltage. Vpp is usually 12 Volts to the flash chip. Because not all flash chips require this voltage, the member is included only if SOCKET_12_VOLTS is defined.

Vpp must be known to be good on exit and is assumed to be ON constantly in RFA environments.

**NOTE:** The macro SOCKET_12_VOLTS is only alterable by users that have source code for the TrueFFS core.

**rfaVppOff()**

TrueFFS calls this routine to turn off a programming voltage (Vpp, usually 12 Volts) to the flash chip. Because not all flash chips require this voltage, the member is included only if SOCKET_12_VOLTS is defined. This routine is not optional, and must always be implemented. Vpp is assumed to be ON constantly in RFA environments.

**rfaSocketInit()**

TrueFFS calls this routine before it tries to access the socket. TrueFFS uses this routine to handle any initialization that is necessary before accessing the socket, especially if that initialization was not possible at socket registration time. For example, if no hardware detection was performed at socket registration time, or if the flash memory medium is removable, this routine should detect the flash memory medium and respond appropriately, including setting cardDetected to FALSE if it is missing.

**rfaSetWindow()**

TrueFFS uses window.base to store the base address of the memory window on the flash memory, and window.size to store the size of the memory window. TrueFFS assumes that it has exclusive access to the window. That is, after it sets one of these window characteristics, it does not expect your application to directly change any of them, and could crash if you do. An exception to this is the mapping register. Because TrueFFS always reestablishes this register when it accesses flash memory, your application may map the window for purposes other than TrueFFS. However, do not do this from an interrupt routine.
rfaSetMappingContext( )

TrueFFS calls this routine to set the window mapping register. This routine performs the sliding action by setting the mapping register to an appropriate value. Therefore, this routine is meaningful only in environments such as PCMCIA, that use the sliding window mechanism to view flash memory. Flash cards in the PCMCIA slot use this routine to access or set a mapping register that moves the effective flash address into the host’s memory window. The mapping process takes a “card address”, an offset in flash, and produces a real address from it. It also wraps the address around to the start of flash if the offset exceeds flash length. The latter is the only reason why the flash size is a required entity in the socket driver. On entry to setMappingContext, vol.window.currentPage is the page already mapped into the window (meaning that it was mapped in by the last call to setMappingContext).

rfaGetAndClearChangeIndicator( )

This routine reads the hardware card-change indication and clears it. It serves as a basis for detecting media-change events. If you have no such hardware capability, return FALSE for this routine (set this function pointer to NULL).

rfaWriteProtected( )

TrueFFS can call this routine to get the current state of the media’s write-protect switch (if available). This routine returns the write-protect state of the media, if available, and always returns FALSE for RFA environments. For more information, see the VxWorks Kernel Programmer’s Guide: Flash File System Support with TrueFFS.

Socket Windowing and Address Mapping

The FLSocket structure (defined in installDir/vxworks-6.x/target/h/tffs/flsocket.h) contains an internal window state structure. If you are porting the socket driver, the following background information about this window structure may be useful when implementing the xxxSetWindow( ) and xxxSetMappingContext( ) routines.

The concept of windowing derives from the PCMCIA world, which formulated the idea of a Host Bus Adapter. The host could allow one of the following situations to exist:

- The PCMCIA bus could be entirely visible in the host’s address range.
- Only a segment of the PCMCIA address range could be visible in the host’s address space.
- Only a segment of the host’s address space could be visible to the PCMCIA.
To support these concepts, PCMCIA specified the use of a “window base register” that may be altered to adjust the view from the window. In typical RFA scenarios, where the device logic is NOR, the window size is that of the amount of flash on the board. In the PCMCIA situation, the window size is implementation-specific. The book *PCMCIA Systems Architecture* by Don Anderson provides an explanation of this concept, with illustrations.

### 6.3.4 Flash Translation Layer

This section provides a detailed discussion of the flash translation layer (FTL).

**Terminology**

Due to the complex nature of the FTL layer, you may find the following definitions useful as a reference for the remainder of this section.

*virtual sector number*

The virtual sector number is what the upper software layers see as a sector number. The virtual sector number is used to reference sectors the upper software layers want to access. The virtual sector number can be decoded into page number and sector in page. These decoded numbers are used to find the logical sector number.

*virtual sector address*

The virtual sector address is the virtual sector number shifted left by 9. This gives a byte offset into the flash array corresponding to a 512 byte sector size. A block allocation map (BAM) entry can contain the virtual sector address for logical sectors that contain data.
The data the virtual sector number references.

**logical sector number**

The logical sector number can be decoded to determine the physical sector address. It decodes to the logical unit number and the sector in unit.

**logical sector address**

The logical sector address is the logical sector number shifted left by 9 (512 byte sectors). This address is used for the virtual block map (VBM) entries.

**logical sector**

If the logical sector term is used, it is referring to the physical sector.
physical sector address
The physical sector address is the memory address in the flash volume where the data that is being referenced by a virtual sector number resides.

physical sector
The physical sector is the data that a virtual sector number references.

garbage sector
This is a physical sector that has been marked as garbage in the BAM. This garbage sector becomes a free sector when a unit transfer happens on the erase unit that contains this garbage sector.

sector in unit
Sector in unit is the number of sectors into the physical erase unit where the physical sector resides.

logical unit number
The logical unit number refers to the number assigned to a physical erase unit when it becomes part of a flash volume. This logical unit number is located in the erase unit header. Transfer units do not have a logical unit number assigned to them. The logical unit number is used to index into the vol.logicalUnits[ ] array. The vol.logicalUnits[ ] array contains pointers into the vol.physicalUnits[ ] array.

logical erase unit
Not all erase units are logical erase units. Only erase units that have been assigned a logical unit number are logical erase units.

physical unit number
The physical unit number can be used to traverse the vol.physicalUnits[ ] array. Generally, the physical unit number is generated when using a logical unit number along with the vol.logicalUnits[ ] array and then using pointer arithmetic off of the pointer into the vol.physicalUnits[ ] array.

physical erase unit
This is the same as an erase unit.

erase unit
An erase unit is the smallest area on the flash device that can be erased. Each erase unit in the flash volume is divided into several physical sectors and includes a header called an erase unit header.

erase unit header
The erase unit header contains information about the FLT volume. It also contains some current information about the erase unit, such as wear-leveling and logical unit number.
**virtual block map**

A virtual block map (VBM) table is a map of virtual sector numbers to logical sector addresses. Each VBM page takes up one physical sector. The logical sector number of the physical sector that contains a VBM page is stored in the `vol.pageTable[]` array at the time the flash volume is mounted. A virtual sector number is decoded into a page number to be used with the `vol.pageTable[]` array and the sector in page. Each VBM entry contains one of the following: a logical sector address or a designated free sector or deleted sector.

**block allocation map**

There is a block allocation map (BAM) in each logical erase unit. The BAM starts immediately after the erase unit header. There is a BAM entry for each sector in the erase unit. Each physical sector in the erase unit is labeled in the BAM based on the type of data stored in that sector. The lower 7 bits of each BAM entry is the block allocation type, which states what type of data is located in the physical sector associated with the BAM entry.

**block allocation type**

Block allocation type is a number specifying what type of data is stored in the physical sector associated with the BAM entry.
page table
Page table is synonymous with the `vol.pageTable[]` array. This array contains each VBM page and some directly addressable sectors. It has logical sector numbers to these VBM pages and directly addressable sectors.

page number
Page number is an index into the `vol.pageTable[]` array. This page number can be generated using the virtual sector number. When the page number is used with the `vol.pageTable[]` array, it finds the logical sector number of the VBM page.

sector in page
After the page number is used to get the logical sector number of the VBM page, the VBM entry located at the sector in page is used to find the logical sector address of virtual sector number in question.

replacement page
The replacement page is used to increase the speed of the FTL. When a VBM entry has been used and the entry needs to be replaced, a replacement page is used for the new entry. In this manner, a new VBM page is not needed every time an entry must be overwritten.

transfer unit
When a unit gets full and has garbage sectors, it can be transferred to a clean erase unit. This clean erase unit is called a transfer unit. After the transfer, the transfer unit takes on the logical unit number of the erase unit that was transferred. The old erase unit is then erased.

unit transfer
Unit transfer refers to the act of transferring valid data from a logical unit to a transfer unit.

directly addressable sectors
Directly addressable sectors are virtual sectors that are not mapped through the VBM pages. These sectors are added to the end of the `vol.pageTable[]` array for direct access. These directly addressable sectors are set up to be the beginning of the virtual disk. Because the FAT is at the beginning of the virtual disk, this results in a modest speed increase. All sectors can be set up to go through this direct addressing. However, this results in the `vol.pageTable[]` taking up additional physical memory.

flash volume
The flash volume is composed of all erase units associated with the flash device, including reserved flash.
Overview

Flash can only be written once before it must be erased. Because an erase unit is larger than a sector, FTL cannot erase a single sector. Instead, it must erase an entire erase unit. This is the primary reasons for having a flash translation layer. In addition to this primary responsibility, the FTL is also responsible for wear-leveling.

The FTL keeps track of sectors and their physical sector addresses through several tables stored on the flash device. Virtual sectors are mapped to logical sectors through these tables. Virtual sectors are what the upper software layers use to reference their sectors. Logical sectors are what the FTL uses to find the physical sector. Virtual sectors map to logical sectors, while logical sectors map to physical sectors.

As an example, the FTL is needed when a sector is being overwritten. In this case, the virtual sector requires a new physical sector to store this new data because it cannot write over the old data. So, a new logical sector, referencing the new physical sector, is allocated. The data is then written to this new physical sector. The new logical sector is mapped to the virtual sector in question. The old logical sector is then deleted by marking it as a garbage sector. Marking as a garbage sector allows the old physical sector to be reclaimed at a later time, as part of garbage collection.

Special care must be taken to reclaim garbage sectors. This reclamation of garbage sectors is done by transferring valid data from an erase unit to an erase unit that has already been erased. This erased erase unit is called a transfer unit.

As mentioned previously, in order to erase even a single bit of flash, an entire erase unit must be erased. (For more information on erase units, see Erase Units, p.178.) The flash translation layer controls erasing and the movement of data around the flash device.

Structures

The Flare structure contains information about the physical device. Before more detail of the Flare structure is shown, some type defines and other structures must be explained.
typedef unsigned long SectorNo;
typedef long int LogicalAddress;
typedef long int VirtualAddress;
typedef SectorNo LogicalSectorNo;
typedef SectorNo VirtualSectorNo;
typedef unsigned short UnitNo;
typedef unsigned long CardAddress;

Each of the typedefs serves a particular purpose. The type defines are as follows:

**SectorNo**

- **SectorNo** is used for both virtual and logical sector numbers. It can be type defined differently based on the maximum volume size. If the maximum volume size is 32 MB or less, **SectorNo** is defined as an unsigned short. If the maximum volume size is more than 32 MB, **SectorNo** is defined as an unsigned long. All versions of VxWorks have the maximum volume size set greater than 32 MB.

**LogicalSectorNo**

- **LogicalSectorNo** is used for logical sector numbers.

**VirtualSectorNo**

- **VirtualSectorNo** is used for virtual sector number.

**LogicalAddress**

- **LogicalAddress** is used for logical sector addresses.

**VirtualAddress**

- **VirtualAddress** is used for virtual sector addresses.

**UnitNo**

- **UnitNo** is used to store logical unit numbers.

**CardAddress**

- **CardAddress** is used to store the physical address of the flash in question. This address does not need to align with anything.

**NOTE:** Unfortunately, the above type defines, when used in the source code, do not always follow the names described. For example, there are several places where a **VirtualSectorNo** holds a virtual sector address.

There is a structure called **Unit** that is used to reference units in the **Flare** structure.

typedef struct
{
    short noOfFreeSectors;
    short noOfGarbageSectors;
} Unit;
The Flare structure is really the structure TLrec therefore both structures are declared. This Flare structure is used in ftllite.c in the vols[ ] array.

typedef TLrec Flare;
struct TLrec {
    FLBoolean badFormat; /* true if FTL format is bad */
    VirtualSectorNo totalFreeSectors; /* Free sectors on volume */
    SectorNo virtualSectors; /* size of virtual volume */
    unsigned int unitSizeBits; /* log2 of unit size */
    unsigned int erasableBlockSizeBits; /* log2 of erasable block size */
    UnitNo noOfUnits;
    UnitNo noOfTransferUnits;
    UnitNo firstPhysicalEUN;
    int noOfPages;
    VirtualSectorNo directAddressingSectors; /* no. of directly addressable sectors */
    VirtualAddress directAddressingMemory; /* end of directly addressable memory */
    CardAddress unitOffsetTable; /* = 1 << unitSizeBits - 1 */
    unsigned int sectorsPerUnit;
    unsigned int unitHeaderSectors; /* sectors used by unit header */
    Unit * physicalUnits; /* unit table by physical no. */
    Unit ** logicalUnits; /* unit table by logical no. */
    Unit * transferUnit; /* The active transfer unit */
    LogicalSectorNo * pageTable; /* page translation table */
    LogicalSectorNo replacementPageAddress;
    VirtualSectorNo replacementPageNo;
    SectorNo mappedSectorNo;
    CardAddress mappedSectorAddress;
    unsigned long currWearLevelingInfo;
    #ifdef BACKGROUND
        Unit * unitEraseInProgress; /* Unit currently being formatted */
        FLStatus garbageCollectStatus; /* Status of garbage collection */
        long int mirrorOffset;
        CardAddress mirrorFrom, mirrorTo;
    #endif
    #ifndef SINGLE_BUFFER
        FLBuffer * volBuffer; /* Define a sector buffer */
    #endif
    #ifndef MALLOC_TFFS
        char heap[HEAP_SIZE];
    #endif
};
Each field is defined as described below. These values represent the whole flash volume.

**badFormat**
If the flash volume does not mount correctly, `badFormat` is set to `TRUE`. Several routines check this value before continuing.

**totalFreeSectors**
The number of physical sectors in the volume that are set to `FREE_SECTOR` in the BAM tables. This is not the number of physical sectors not in use, because this would also include garbage sectors in the total.

**virtualSectors**
`virtualSectors` is the number of physical sectors available to the upper software layers. This is not the total number of physical sectors. Physical sectors that do not get counted are physical sectors in transfer units, physical sectors that are part of the FTL control structures (such as the BAM and erase unit headers), and physical sectors that are used by the VBM pages. Other physical sectors that are not considered are those that are reserved for the FTL layer to increase its efficiency. These reserved physical sectors are user defined with `percentUse` in the `tffsFormatParms` structure passed when formatting the flash volume.

```c
vol.sectorsPerUnit = (1 << (vol.unitSizeBits - SECTOR_SIZE_BITS));
vol.unitHeaderSectors = (((vol.bamOffset + sizeof(VirtualAddress) * vol.sectorPerUnit - 1) >> SECTOR_SIZE_BITS) + 1);
vol.virtualSectors = ((vol.noOfUnits - vol.firstPhysicalEUN – formatParams->noOfSpareUnits) *
                      (vol.sectorsPerUnit - vol.unitHeaderSectors) *
                      formatParams->percentUse / 100) –
                      (vol.noOfPages + 1);
```

**unitSizeBits**
`unitSizeBits` is the number of bits it takes to store the erase unit size. The outcome of the calculation is \( \ln(\text{vol.flash.erasableBlockSize}) / \ln(2) \).

```c
unitSizeBits = vol.erasableBlockSizeBits;
```

**erasableBlockSizeBits**
`erasableBlockSizeBits` is the number of bits it takes to store the erasable block size. This erasable block size is the same as erase unit size. Because the size of an erase unit is flash-device dependent, there is no quick calculation. The `initFTL()` routine uses an optimized routine to calculate `vol.erasableBlockSizeBits` through `vol.flash.erasableBlockSize`. The size of the erase unit, `vol.flash.erasableBlockSize`, is setup by the MTD. The outcome of the calculation is \( \ln(\text{vol.flash.erasableBlockSize}) / \ln(2) \).
noOfUnits

noOfUnits is the total number of units in the volume, including transfer units, and reserved units. Do not confuse Flare structure noOfUnits with the unit headers of the noOfUnits field. These values are different.

\[
\text{noOfUnits} = ((\text{vol.flash.noOfChips} \times \text{vol.flash.chipSize}) >> \text{vol.unitSizeBits});
\]

noOfTransferUnits

noOfTransferUnits is the number of transfer units on the flash volume. This value is user defined when the flash volume is formatted as noOfSpareUnits in the structure tffsFormatParams.

firstPhysicalEUN

firstPhysicalEUN is the physical unit number of the file system on the flash volume. This number is necessary because some of the flash volume can be reserved for use by the user.

\[
\text{firstPhysicalEUN} = (((\text{formatParams->bootImageLen} - 1) >> \text{vol.unitSizeBits}) + 1);
\]

noOfPages

noOfPages is the number of VBM pages needed to map all of the virtual sectors that the upper software layers can access.

\[
\text{noOfPages} = ((\text{vol.virtualSectors} \times \text{SECTOR\_SIZE} - 1) >> \text{PAGE\_SIZE\_BITS}) + 1;
\]

directAddressingSectors

directAddressingSectors are logical sectors that are directly mapped. These logical sectors are not mapped through the VBM pages. Since VBM pages are not virtual sectors, they too are part of this number. The number of virtual sectors that are part of directAddressingSectors is based on a user defined value when the flash volume is formatted. This user defined value is vmAddressingLimit in the structure tffsFormatParams.

\[
\text{directAddressingSectors} = (\text{formatParams->vmAddressingLimit} / \text{SECTOR\_SIZE}) + \text{vol.noOfPages};
\]

directAddressingMemory

directAddressingMemory is the amount of flash that is directly mapped through the pageTable[ ] array and not through the VBM pages. This increases the speed to these virtual sectors. This value is user defined when the flash volume is formatted as vmAddressingLimit in the structure tffsFormatParams.

\[
\text{directAddressingMemory} = \text{formatParams->vmAddressingLimit};
\]
unitOffsetMask

unitOffsetMask is used in one calculation in the source code. The routine logical2Physical() is used to calculate the physical sector address of a logical sector.

\[
\text{unitOffsetMask} = (1L << \text{vol.unitSizeBits}) - 1;
\]

bamOffset

bamOffset is the offset of the BAM with reference to the beginning of the erase unit header. Typically bamOffset is just sizeof(UnitHeader) which is 0x44 (68). If embeddedCIS of formatParams is anything but 0, the calculation becomes difficult.

\[
\text{bamOffset} = \text{sizeof(UnitHeader)};
\]

or this if formatParams->embeddedCISLength is not 0

\[
\text{bamOffset} = \text{sizeof(UnitHeader)} - (\text{sizeof uh->embeddedCIS}) + (\text{formatParams->embeddedCISLength} + 3) / 4 * 4;
\]

sectorsPerUnit

sectorsPerUnit is the number of physical sectors in an erase unit.

\[
\text{sectorsPerUnit} = (1 << (\text{vol.unitSizeBits} - \text{SECTOR_SIZE_BITS}));
\]

unitHeaderSectors

unitHeaderSectors is the number of physical sectors that are taken up by the erase unit header and the BAM table for a single erase unit. Note that allocEntryOffset() returns offset of the BAM entry from the beginning of the erase unit.

\[
\text{unitHeaderSectors} = ((\text{allocEntryOffset}(&\text{vol.vol.sectorsPerUnit}) - 1) >> \text{SECTOR_SIZE_BITS}) + 1;
\]

physicalUnits

physicalUnits[] array stores information about each physical unit. This information is the simple structure called Unit (defined earlier). It is filled out when the flash volume is mounted. &physicalUnits[x] is passed into many routines, which is used to find the index into the physicalUnits[] array. The index into the physicalUnits[] array is the physical unit number.

logicalUnits

logicalUnits[] array stores pointers to an index into the physicalUnits[] array. The logicalUnits[] array index is the logical unit number. This is used when converting from logical unit numbers to physical unit numbers. logicalUnits[x] maps to &physicalUnits[y]. The physical unit number can be determined using pointer arithmetic off of &physicalUnits[y].
transferUnit
  transferUnit is the pointer &physicalUnits[x] where x is the physical unit number of the transfer unit.

pageTable
  pageTable[ ] array is a list of logical sector numbers that reference the directly addressable sectors and the VBM pages. pageTable[0] up to pageTable[noOfPages - 1] are VBM pages, while pageTable[noOfPages] to pageTable[directAddressingSectors – 1] are part of the directly addressable memory.

replacementPageAddress
  The logical sector number of the replacement page. When browsing the replacementPageAddress get assigned sectorAddress. sectorAddress is not a logical sector address, but a logical sector number.

replacementPageNo
  replacementPageNo is the VBM page number that the replacement page is replacing.

mappedSectorNo
  mappedSectorNo is the logical sector number that is mapped to a global buffer.

mappedSector
  mappedSector is the physical address of the mapped sector.

mappedSectorAddress
  mappedSectorAddress is the physical sector address of mappedSectorNo.

currWearLevelingInfo
  currWearLevelingInfo is the wear-leveling information about the whole volume. This number is incremented every time an erase unit is erased.

unitEraseInProgress
  Not used. For background garbage collection only.

garbageCollectStatus
  Not used. For background garbage collection only.

mirrorOffset
  Not used. For background garbage collection only.

mirrorFrom
  Not used. For background garbage collection only.

mirrorTo
  Not used. For background garbage collection only.
volBuffer

volBuffer is a pointer to a buffer that is used by ALL vol. structures. This buffer is defined as the structure FLBuffer. This buffer can hold the data of a single sector.

flash

Structure to get access to the flash primitives. This is accessible to the MTD.

heap[HEAP_SIZE]

Not used. For operating systems that do not support malloc().

Erase Units

In VxWorks, an erase unit is the smallest area that can be erased on the flash device. Typical sizes for an erase unit are 64 KB and 128 KB. The size of an erase unit depends on the type of flash chips used, and if they are interleaved together.

Interleaved Flash Chips

Flash chips are sometimes interleaved to allow larger word access to flash. That is, two 8-bit flash chips can be interleaved together to allow 16-bit access. When working with interleaved flash, the smallest area that can be erased is increased. This smallest erasable area is called an erase unit.

The minimum erase area is increased by a factor of the number of flash chips interleaved together. In the case of two 8-bit flash chips interleaved to allow 16-bit access to the flash device, two erase blocks would make up one erase unit. Therefore, in this case, if the erase block size is 64 KB, the erase unit size is 128 KB.

A flash device is split up into erase units, while the erase units are sub divided into physical sectors. These physical sectors are sometimes called read/write blocks in other documentation. A physical sector is 512 bytes in size and is hard coded into the FTL.

NOTE: As a note, erasing an erase unit is sometimes called formatting. This is not the same as a DOS format.
An erase unit is also split up into 3 other sections. These sections overlay the physical sectors. These 3 sections are the erase unit header, block allocation map (BAM), and the data area. The BAM starts immediately after the erase unit header, and may not align on a physical sector boundary. The data area always aligns with a physical sector.

**Erase Unit Header**

The erase unit header contains data about the flash volume and the current erase unit. The erase unit header is defined as `UnitHeader` in `fltlite.c`. Note that the erase unit header is only a part of the erase units that are part of the file system. The erase unit header is *not* part of the erase units in the user reserved area.
typedef struct
{
    char formatPattern[15];
    unsigned char noOfTransferUnits; /* no. of transfer units */
    LBulong wearLevelingInfo;
    LBushort logicalUnitNo;
    unsigned char log2SectorSize;
    unsigned char log2UnitSize;
    LBushort firstPhysicalEUN; /* units reserved for boot image */
    LBushort noOfUnits; /* no. of formatted units */
    LBulong virtualMediumSize; /* virtual size of volume */
    LBulong directAddressingMemory; /* directly addressable memory */
    LBushort noOfPages; /* no. of virtual pages */
    unsigned char flags;
    unsigned char eccCode;
    LBulong serialNumber;
    LBulong altEUHoffset;
    LBulong BAMoffset;
    char reserved[12];
    char embeddedCIS[4]; /* Actual length may be larger. By default, this contains FF’s */
} UnitHeader;

A detailed description of the parameters in the erase unit header follows this introduction. The following description also includes some calculations that require other structures. formatParams, which is described in a previous section, is used. Information used from the Flare structure is accessed through vol. For more information on the Flare structure, see Structures, p.171.

formatPattern

formatPattern consists of the PCMCIA link target tuple (first 5 bytes) and the PCMCIA data organization tuple (last 10 bytes). This formatPattern in the erase unit header is used to verify that the erase unit is valid by calling the verifyFormat() routine. FORMAT_PATTERN is defined in ftllite.c as a comparison string used by verifyFormat(). The first 2 bytes and the 6th byte are ignored by verifyFormat().

static char FORMAT_PATTERN[15] = {0x13, 3, 'C', 'I', 'S',
                                  0x46, 57, 0, 'F', 'T', 'L', '1', '0', '0', 0};

noOfTransferUnits

The noOfTransferUnits is set by the user when the flash volume is formatted. For more information, refer to noOfSpareUnits in tffsFormatParams.

noOfTransferUnits = formatParams->noOfSpareUnits;
wearLevelingInfo

`wearLevelingInfo` is used to keep track wear-leveling for the flash volume. This parameter is specific to each erase unit. This allows the FTL to keep track of the order in which erase units are erased. When an erase unit is erased, `wearLevelingInfo` increments the value of `vol.currWearLevelingInfo`. `vol.currWearLevelingInfo` keeps track of the number of times any of the erase units have been erased on a flash volume.

logicalUnitNo

`logicalUnitNo` is the logical unit number of an erase unit. This number is unique for each logical erase unit. `logicalUnitNo` is used as an index into `vol.logicalUnits[ ]` array. When mounting a flash volume, the physical unit associated with `vol.physicalUnits[ ]` is assigned to `vol.logicalUnits[logicalUnitNo]`. If the erase unit is not mapped into the `vol.logicalUnits[ ]` array, `logicalUnitNo` has a value of `UNASSIGNED_UNIT_NO` (0xffff) or `MARKED_FOR_ERASE` (0x7fff). If the value is `UNASSIGNED_UNIT_NO` the erase unit is a transfer unit. If the value is `MARKED_FOR_ERASE` then the erase unit is in the process of a unit transfer. For a very short period of time during a unit transfer there can be 2 erase units with the same `logicalUnitNo`.

log2SectorSize

`log2SectorSize` is the number of bits needed to store the physical sector size. Because the sector size is 512 bytes, the calculation is `(ln(512) / ln(2))` which is 9.

```c
#define SECTOR_SIZE_BITS 9
log2SectorSize = SECTOR_SIZE_BITS;
```

log2UnitSize

Similar to the calculation of `log2SectorSize`, but instead of using the size of a physical sector, it uses the size of an erase unit. Because the size of an erase unit is flash device dependent, there is no quick calculation. The `initFTL()` uses an optimized routine to calculate `vol.erasableBlockSizeBits` through `vol.flash.erasableBlockSize`. The `vol.flash.erasableBlockSize`, which is the size of the erase unit, is set up by the MTD. The end result is that `log2UnitSize` is set to `(ln(vol.flash.erasableBlockSize) / ln(2))`.

```c
vol.unitSizeBits = vol.erasableBlockSizeBits;
log2UnitSize = vol.unitSizeBits;
```

firstPhysicalEUN

`firstPhysicalEUN` is the physical unit number of the file system on the flash volume. This number is necessary since some of the flash volume can be reserved for user use. `vol.firstPhysicalEUN` is first calculated during a format of the FTL partition.
vol.firstPhysicalEUN = (((formatParams->bootImageLen - 1) >> vol.unitSizeBits) + 1);

noOfUnits

noOfUnits is the number of erase units used by the file system. Note that this number may not include all of the erase units on the flash volume, since reserved flash is not counted. This calculation uses vol.noOfUnits, which is not the same as the erase unit header's noOfUnits. vol.noOfUnits is the total number of erase units for the flash volume.

vol.noOfUnits = ((vol.flash.noOfChips * vol.flash.chipSize) >> vol.unitSizeBits);

virtualMediumSize

virtualMediumSize is the total number of physical sectors that can be used by the file system, and thus the number of sectors available to the upper software layers. This is sometimes called the formatted size. This calculation is very complex, since it has to exclude transfer units, BAM, erase unit headers, page tables, and so forth.

vol.sectorsPerUnit = (1 << (vol.unitSizeBits - SECTOR_SIZE_BITS));
vol.unitHeaderSectors = (((vol.bamOffset + sizeof(VirtualAddress) * vol.sectorsPerUnit - 1) >> SECTOR_SIZE_BITS) + 1);
vol.virtualSectors = ((vol.noOfUnits - vol.firstPhysicalEUN - formatParams->noOfSpareUnits) * (vol.sectorsPerUnit - vol.unitHeaderSectors) * formatParams->percentUse / 100) - (vol.noOfPages + 1);

#define SECTOR_SIZE 512
virtualMediumSize = (vol.virtualSectors * SECTOR_SIZE);

directAddressingMemory

directAddressingMemory is based off of vmAddressingLimit from the tffsFormatParams structure. vmAddressingLimit has already been discussed.

vol.directAddressingMemory = formatParams->vmAddressingLimit;
directAddressingMemory = vol.directAddressingMemory;

noOfPages

noOfPages is the number of physical sectors needed for all the VBM pages. Part of this calculation is somewhat confusing. Because each entry in a VBM page is 4 bytes in size, (SECTOR_SIZE_BITS - 2) is used. A simpler calculation would be ((vol.virtualSectors - 1) >> (SECTOR_SIZE_BITS - 2) + 1) rather than what is used in the source code.

#define PAGE_SIZE_BITS (SECTOR_SIZE_BITS + (SECTOR_SIZE_BITS - 2))
vol.noOfPages = ((vol.virtualSectors * SECTOR_SIZE - 1) >> PAGE_SIZE_BITS) + 1;

noOfPages = vol.noOfPages;
flags
flags is not used and is set to 0 during the format of the flash volume. If any other value is found in this field, the erase unit is considered bad by the software.

eccCode
eccCode is the Error Detection and Correction (EDAC) type. Set to 0xff as the default value. Must be either 0xff or 0x00 or the erase unit is considered bad.

serialNumber
This field is not used, and is zeroed out.

altEUHOffset
This is the offset to the alternate erase unit header. Note that this value is not used in any calculation. altEUHOffset is set to 0 during the format of the flash volume.

BAMoffset
BAMoffset is the offset of the BAM with reference to the beginning of the erase unit header. Typically BAMoffset is just sizeof(UnitHeader) which is 0x44 (68). If embeddedCIS of formatParams is anything but 0, the calculation becomes difficult.

\[
\text{vol.bamOffset} = \text{sizeof(UnitHeader)};
\]

or this if formatParams->embeddedCIS is not 0

\[
\text{vol.bamOffset} = \text{sizeof(UnitHeader)} - (\text{sizeof uh->embeddedCIS}) + (\text{formatParams->embeddedCISlength + 3}) / 4 * 4;
\]

BAMoffset = vol.bamOffset;

reserved
12 bytes. Reserved for future use.

embeddedCIS
embeddedCIS is the location of the embedded CIS information. The default size is 4 bytes, which the user can override.

The above fields are derived when the flash volume is formatted. All of these fields are the same for each erase unit in the flash volume except wearLevelingInfo and logicalUnitNo which are specific to an erase unit. When an erase unit is being formatted (or erased), all of these fields are copied from another valid erase unit, except wearLevelingInfo and logicalUnitNo.
7.1 Introduction 185
7.2 SCSI Overview 186
7.3 SCSI BSP Interface 224
7.4 The SCSI Driver Development Process 226
7.5 Common SCSI Driver Development Issues 227

7.1 Introduction

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

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

You must first understand the basic functionality of each of these components before you can extend the functionality of the SCSI libraries or add new SCSI controller drivers. To help you gain that understanding, this chapter describes the general layout of the various SCSI modules, discusses the internals of the SCSI libraries (and their programming interface with the SCSI controller drivers), and describes the process of developing a controller-specific SCSI driver.
When a VxWorks task requests SCSI service by invoking a SCSI library routine such as `scsiInquiry()` . Since we are assuming a SCSI-2 configuration, first the `scsi2Inquiry()` routine is invoked which in turn invokes `scsiTransact()` (see Forming SCSI Commands, p.189). `scsiTransact()` invokes `scsiCommand()` , the routine that allocates a SCSI thread, executes the thread, and then deletes it.

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

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

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

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

## 7.2 SCSI Overview

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

### 7.2.1 Layout of SCSI Modules

Figure 7-1 shows all the SCSI library modules and the relationship between them and several typical drivers. The SCSI libraries contain a variety of data structures. The important data structures and their relationships are described in the
following subsections. The general design of the data structures is object-oriented; data structures represent real and abstract SCSI objects such as peripheral devices, controllers, and block devices.

**SCSI Objects and Data Structures**

Figure 7-2 illustrates the relationship between the various physical and logical SCSI objects and the corresponding data structures.
Figure 7-3 illustrates the contents of these data structures and their relationships in more detail.

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

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

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

**SCSI_PHYS_DEV**
This structure contains information about available logical devices plus information about the various threads.
SEQ_DEV
This structure represents a sequential logical device such as a tape drive.

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

Figure 7-3 Controller- and Driver-Specific Data Structures

**Forming SCSI Commands**

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

The `scsiTransact()` routine is the general routine in `scsi2Lib` that handles processing of all SCSI commands originating in `scsiDirectLib`, `scsiCommonLib`, etc.
and `scsiSeqLib`. This paradigm should be used to extend SCSI library support to other device classes (`scsiXXXLib`).

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

### 7.2.2 The VxWorks OS Interface

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

#### Libraries

This section describes the following libraries:

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

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

#### SCSI Manager (`scsiMgrLib`)

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

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

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

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

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

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

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

**Limitations**

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

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

191
the maximum number of time-outs that can be put on the time-out queue (scsiMgrTimeoutQSize)
- time-out values.

Configuration

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

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

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

SCSI Controller Library (scsiCtrlLib)

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

There is a well defined interface between the SCSI libraries and the controller driver of such drivers, and this interface is defined in *Driver Programming Interface*, p.193.

SCSI Direct Access Library (scsiDirectLib)

The SCSI direct access library scsiDirectLib encapsulates all the routines that implement the SCSI direct access commands as defined in the **SCSI ANSI**
In addition to all the direct access commands, `scsiDirectLib` provides the routines that supply the `BLK_DEV` abstraction for SCSI direct access peripheral devices.

**SCSI Sequential Access Library (scsiSeqLib)**

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

**SCSI Common Access Library (scsiCommonLib)**

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

**Driver Programming Interface**

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

**Basic SCSI Controller Driver**

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

**The Programming Interface**

A well-defined programming interface exists between the controller driver of any basic SCSI controller and the SCSI libraries. Every basic controller driver must provide the following routines to the SCSI libraries:
xxxDevSelect()  
This routine selects a SCSI peripheral device with the attention (ATN) signal asserted.

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

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

xxxXferParamsSet()  
This routine sets the synchronous data transfer parameters on the SCSI controller.

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

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

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

scsiMgrEventNotify()  
This routine notifies the SCSI manager of a SCSI event that has occurred. Events are defined in scsi2Lib.h. However, more events can be defined by the controller driver, and events can also be bundled by the driver. In this case, the SCSI_CTRL field scsiEventProc must be set to this driver-specific routine during driver initialization.

A Template Driver

The following example shows a template for a basic SCSI controller driver, without any specific hardware constraints. The basic structure of the driver is like any other VxWorks driver. The main routines consist of the following:

- A xxxCtrlCreate() routine, that is invoked from the BSP routine sysScsiInit() located in the BSP file sysScsi.c.
- An ISR called xxxIntr() that handles all the interrupts, deciphers what SCSI event has occurred, and passes that event information to the SCSI manager via the scsiMgrEventNotify() routine.
The SCSI libraries instruct the driver via the xxxDevSelect() and xxxInfoXfer() routines, and the controller driver communicates back to the libraries by means of the scsiMgrEventNotify() routine.

Example 7-1  Basic SCSI Controller Driver

/* xxxLib.c - XXX SCSI-Bus Interface Controller library (SCSI-2) */

/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/* modification history
---------------------
01a,12sep96,dds  written */

/* DESCRIPTION
This library contains part of the I/O driver for the XXX family of SCSI-2 
Bus Interface Controllers (SBIC). It is designed to work with scsi2Lib.
The driver routines in this library depend on the SCSI-2 ANSI specification;
for general driver routines and for overall SBIC documentation, see xxxLib.

INCLUDE FILES
xxx.h

SEE ALSO: scsiLib, scsi2Lib,
the VxWorks programmer’s guides */

#include "vxWorks.h"
#include "drv/scsi/xxx.h"

typedef XXX_SCSI_CTRL SBIC; /* SBIC: SCSI Bus Interface Controller struct */

/* globals */

int xxxXferDoneSemOptions = SEM_Q_PRIORITY;
char *xxxScsiTaskName = SCSI_DEF_TASK_NAME;

IMPORT SCSI_CTRL *pSysScsiCtrl;

/**************************
* xxxCtrlCreate - create and partially initialize a SCSI controller structure
* This routine creates a SCSI controller data structure and must be called
* before using a SCSI controller chip. It should be called once and only
* once for a specified SCSI controller. Since it allocates memory for a
* structure needed by all routines in xxxLib, it must be called before
* any other routines in the library.
* After calling this routine, at least one call to xxxCtrl1Init() should
* be made before any SCSI transaction is initiated using the SCSI controller.
  *
  * RETURNS: A pointer to the SCSI controller structure, or NULL if memory is
  * insufficient or parameters are invalid.
  */

XXX_SCSI_CTRL *xxxCtrlCreate
{
  FAST UINT8 *sbicBaseAdrs, /* base address of the SBIC */
  int regOffset, /* address offset between SBIC registers */
  UINT clkPeriod, /* period of the SBIC clock (nsec) */
  FUNCPTR sysScsiBusReset, /* function to reset SCSI bus */
  int sysScsiResetArg, /* argument to pass to above function */
  UINT sysScsiDmaMaxBytes, /* maximum byte count using DMA */
  FUNCPTR sysScsiDmaStart, /* function to start SCSI DMA transfer */
  FUNCPTR sysScsiDmaAbort, /* function to abort SCSI DMA transfer */
  int sysScsiDmaArg /* argument to pass to above functions */
}

FAST SBIC *pSbic; /* ptr to SBIC info */

/* calloc the controller info structure; return NULL if unable */
pSbic = (SBIC *) calloc (1, sizeof (SBIC))

/* Set up sizes of event and thread structures. Must be done before
 * calling "scsiCtrlInit()". */

/* fill in driver-specific routines for scsiLib interface */
pSbic->scsiCtrl.scsiDevSelect = xxxDevSelect;
pSbic->scsiCtrl.scsiInfoXfer = xxxInfoXfer;
pSbic->scsiCtrl.scsiXferParamsQuery = xxxXferParamsQuery;
pSbic->scsiCtrl.scsiXferParamsSet = (FUNCPTR)xxxXferParamsSet;

/* Fill in driver specific variables for scsiLib interface */
pSbic->scsiCtrl.maxBytesPerXfer = sysScsiDmaMaxBytes;

/* fill in generic SCSI info for this controller */
xxxCtrlInit (&pSbic->scsiCtrl);

/* initialize SBIC info transfer synchronization semaphore */
if (semBInit (&pSbic->xferDoneSem, xxxXferDoneSemOptions, SEM_EMPTY) == ERROR)
{
  (void) free ((char *) pSbic);
  return ((XXX_SCSI_CTRL *) NULL);
}

/* initialize state variables */

/* fill in board-specific SCSI bus reset and DMA xfer routines */
/* spawn SCSI manager - use generic code from 'scsiLib.c' */

psBic->scsiCtrl.scsiMgrId = taskSpawn (xxxTaskName,
        xxxTaskPriority,
        xxxTaskOptions,
        xxxTaskStackSize,
        (FUNCPTR) scsiMgr,
        (int) pSbic,
        0, 0, 0, 0, 0, 0, 0, 0);

return (pSbic);

/*************************************************************************
* xxxCtrlInit - initialize a SCSI controller structure
* After a SCSI controller structure is created with xxxCtrlCreate, but
* before using the SCSI controller, it must be initialized by calling this
* routine.
* It may be called more than once if desired. However, it should only be
* called while there is no activity on the SCSI interface.
* RETURNS: OK, or ERROR if out-of-range parameter(s).
*/

LOCAL STATUS xxxCtrlInit
(
    FAST SBIC *pSbic, /* ptr to SBIC info */
    FAST int  scsiCtrlBusId, /* SCSI bus ID of this SBIC */
    FAST UINT defaultSelTimeOut /* default dev. select timeout (microsec) */
)

{ pSbic->scsiCtrl.scsiCtrlBusId = scsiCtrlBusId;

    /* initialize the SBIC hardware */

    xxxHwInit (pSbic);

    return (OK);
}

/*************************************************************************
* xxxHwInit - initialize the SCSI controller to a known state
* This routine puts the SCSI controller into a known quiescent state. It
* does not reset the SCSI bus (and any other devices thereon).
*/

LOCAL void xxxHwInit
(
    SBIC *pSbic /* ptr to an SBIC structure */
)

{ /*
   * Initialize the SCSI controller hardware registers and place the
* chip in a known quiescent state
 */
}

/***************************************************************************
* xxxDevSelect - attempt to select a SCSI device
* RETURNS: OK (no error conditions)
*/

LOCAL STATUS xxxDevSelect
{
    SCSI_CTRL *pScsiCtrl, /* ptr to SCSI controller info */
    int devBusId, /* SCSI bus ID of device to select */
    UINT selTimeOut, /* select t-o period (usec) */
    UINT8 *msgBuf, /* ptr to identification message */
    UINT msgLen, /* maximum number of message bytes */
}
{
    int lockKey; /* saved interrupt lock key */
    lockKey = intLock ();
    /* Select device */
    intUnlock (lockKey);
}

/***************************************************************************
* xxxXferParamsQuery - get (synchronous) transfer parameters
* Updates the synchronous transfer parameters suggested in the call to match
* the SCSI controller's capabilities. Transfer period is in SCSI units
* (multiples of 4 ns).
* RETURNS: OK
*/

LOCAL STATUS xxxXferParamsQuery
{
    SCSI_CTRL *pScsiCtrl, /* ptr to SBIC info */
    UINT8 *pOffset, /* max REQ/ACK offset [in/out] */
    UINT8 *pPeriod /* min transfer period [in/out] */
}
{
    /* read offset and period values */
    return (OK);
}

/***************************************************************************
* xxxXferParamsSet - set transfer parameters
* Programs the SCSI controller to use the specified transfer parameters. An
* offset of zero specifies asynchronous transfer (period is then irrelevant).
LOCAL STATUS xxxXferParamsSet
{
    SCSI_CTRL *pScsiCtrl, /* ptr to SBIC info */
    UINT8 offset, /* max REQ/ACK offset */
    UINT8 period /* min transfer period */
}
{
    /* set the appropriate SCSI controller registers */
    return (OK);
}

/***************************************************************************
* xxxInfoXfer - transfer information bytes to/from target via SCSI bus
* * Executes a "Transfer Info" command to read (write) bytes from (to) the
* * SCSI bus. If the transfer phase is DATA IN or DATA OUT and there is a
* * DMA routine available, DMA is used - otherwise it's a tight programmed
* * i/o loop.
* * * RETURNS: Number of bytes transferred across SCSI bus, or ERROR.
* */
LOCAL int xxxInfoXfer
{
    FAST SCSI_CTRL *pScsiCtrl, /* ptr to SCSI controller info */
    int phase, /* SCSI phase being transferred */
    FAST UINT8 *pBuf, /* ptr to byte buffer for i/o */
    UINT bufLength /* number of bytes to be transferred */
}
{
    pSbic = (SBIC *) pScsiCtrl;
    /* Handle phase changes */
    /* Start DMA, if used, or programmed i/o loop to transfer data */
    /* Wait for transfer to complete: find out how many bytes transferred */
    semTake (&pSbic->xferDoneSem, WAIT_FOREVER);
    /*
    * If there are bytes left to be transferred return ERROR
    * If DMA is used for transfer do a SCSI DMA Abort
    */
    xxxXferCountGet (pSbic, &bytesLeft);
    return (bufLength - bytesLeft);
}
/*****************************************************************************/
* xxxXferCountSet - load the SCSI controller transfer counter with count.
* RETURNS: OK if count is in range 0 - 0xffffff, otherwise ERROR.
*/

LOCAL STATUS xxxXferCountSet
{
    FAST SBIC *pSbic,    /* ptr to SBIC info */
    FAST UINT count     /* count value to load */
}
{ /* set the appropriate SCSI controller registers */
}

/***************************************************************************/
* xxxXferCountGet - fetch the SCSI controller transfer count
* The value of the transfer counter is copied to *pCount.
*/

LOCAL void xxxXferCountGet
{
    FAST SBIC *pSbic,    /* ptr to SBIC info */
    FAST UINT *pCount    /* ptr to returned value */
}
{ /* read the appropriate SCSI controller registers */
}

/***************************************************************************/
* xxxCommand - write a command code to the SCSI controller Command Register
*/

LOCAL void xxxCommand
{
    SBIC *pSbic,        /* ptr to SBIC info */
    UINT8 cmdCode       /* new command code */
}
{ /* set the appropriate SCSI controller registers */
}

/***************************************************************************/
* xxxIntr - interrupt service routine for the SCSI controller
*/

LOCAL void xxxIntr
{
    SBIC *pSbic        /* ptr to SBIC info */
}
{ SCSI_EVENT event;
/* Check the SCSI status. Handle state transitions */

switch (scsiStatus)
{
    ...

    /* the list of event types is defined in scsi2Lib.h */

    case ...
    {
        event.type = SCSI_EVENT_XFER_REQUEST;
        event.phase = busPhase;
        break;
    }

    case ...
}

/* Synchronize with task-level code */

semGive (&pSbic->xferDoneSem);

/* Post event to SCSI manager for further processing */

scsiMgrEventNotify ((SCSI_CTRL *)pSbic, &event, sizeof (event));
}

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

/* xxxRegRead - Get the contents of a specified SCSI controller register */

LOCAL void xxxRegRead
{
    SBIC *pSbic,
    UINT8 regAdrs,
    int *pDatum
    
    { /* read the appropriate SCSI controller registers */

    }

} /* xxxRegWrite - write a value to a specified SCSI controller register */

LOCAL void xxxRegWrite
{
    SBIC *pSbic,
    UINT8 regAdrs,
    UINT8 datum

    { /* write the appropriate SCSI controller registers */

}
Advanced SCSI Controller Driver

The advanced SCSI controller incorporates all the low-level state machine routines within the driver. This functionality replaces that provided by `scsiCtrlLib`. Most advanced SCSI controllers have their own SCSI I/O processor which enhances performance by managing all the low-level activities on the SCSI bus, such as phase changes and DMA data transfers. Usually the instructions to the I/O processor are machine language instructions which are written in a higher level assembly language and compiled into machine instructions. These machine instructions reside in the main DRAM area and are fetched by the I/O processor from DRAM by using a SCSI program counter and some form of indirect addressing.

In the case of advanced SCSI controllers, there is usually additional event information described in a driver-specific structure such as `XXX_EVENT` (where `XXX` refers to the SCSI driver module prefix). Many thread management routines are part of the controller driver, which is not true of the basic SCSI controller drivers.

The Programming Interface

The programming interface between the advanced SCSI controller driver and the SCSI libraries consists of routines that must be supplied by the driver and library routines which are invoked by the driver. The driver routines are not required to conform to the naming convention used here, because the routines are accessed by means of function pointers which are set in the `xxxCtrlCreate()` routine. However, this naming convention is recommended. The routines (or equivalents) that the driver must supply are:

`xxxEventProc()`\(^1\)
This routine is invoked by the SCSI manager to parse events and take appropriate action.

`xxxThreadInit()`
This routine initializes the SCSI thread structures and adds any driver-specific initialization required beyond what is provided by `scsiThreadInit()`.

`xxxThreadActivate()`
This routine activates a SCSI connection, setting the appropriate thread context in the `SCSI_THREAD` data structure and setting all the controller registers with the appropriate values. It may call other driver routines as well as SCSI library routines.

\(^1\) The `xxx` in the routine name is just a place holder for whatever prefix you assign to your SCSI driver module.
xxxThreadAbort()
    If the thread is not actually connected, this routine does nothing. If the thread
    is connected, it sends an ABORT TAG message which causes the SCSI target to
disconnect.

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

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

xxxXferParamsSet()  
This routine sets the synchronous data transfer parameters on the SCSI
controller.

xxxWideXferParamsQuery()  
This routine updates the wide data transfer parameters in the call to match
those of the SCSI controller.

xxxWideXferParamsSet()  
This routine sets the wide data transfer parameters on the SCSI controller.

The advanced controller driver also uses many of the facilities provided by the
SCSI libraries. All the routines invoked by the SCSI controller library can also be
invoked by the driver. Examining the SCSI controller library and the header file
scsi2Lib.h shows all the routines available for the controller driver. The following
list is a typical but not exhaustive list of routines that can be invoked by the driver:

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

scsiMgrEventNotify()  
This routine notifies the SCSI manager of an event that occurred on the SCSI
bus.

scsiWideXferNegotiate()  
This routine initiates or continues wide data transfer negotiation. See the
relevant reference entries and scsi2Lib.h for more details. It is typically
invoked from the xxxThreadActivate() routine.
scsiSyncXferNegotiate()  
This routine initiates or continues synchronous data transfer negotiations. See  
the relevant reference entries and scsi2Lib.h for more details. It is typically  
invoked from the xxxThreadActivate() routine.

scsiMgCtrlEvent()  
This routine sends an event to the SCSI controller state machine. It is usually  
called by the driver xxxEventProc() routine after a selection, re-selection, or  
disconnection.

scsiMgrBusReset()  
This routine resets all physical devices in the SCSI library upon a bus-initiated  
reset. It is typically invoked from xxxEventProc().

scsiMgrThreadEvent()  
This routine sends an event to the thread state machine. It is called by the  
thread management routines within the driver; the entry point to the thread  
routines is by way of xxxEventProc(). In general, xxxEventProc() is the  
general routine which calls other driver-specific thread-management routines.  
For a better understanding, look at the advanced SCSI controller driver  
template and also examine an actual driver.

scsiMsgOutComplete()  
This routine performs post-processing after a SCSI message out has been sent.  
It is also invoked from the driver thread management routines.

scsiMsgInComplete()  
This routine performs post-processing after a SCSI message in is received. It is  
invoked from the driver thread management routines.

scsiMsgOutReject()  
This routine performs post-processing when an outgoing message has been  
rejected.

scsiIdentMsgParse()  
This routine parses an incoming identify message when VxWorks has been  
selected or reselected.

scsiIdentMsgBuild()  
This routine builds an identify message in the caller’s buffer.

scsiCacheSnoopEnable()  
This routine informs the library that hardware cache snooping is enabled and  
that it is unnecessary to call cache-specific routines.
scsiCacheSnoopDisable()

This routine informs the library that hardware snooping has been disabled or does not exist and that the library must perform cache coherency.

scsiCacheSynchronize()

This routine is called by the driver for all cache-coherency needs.

scsiThreadInit()

This routine performs general thread initialization; it is invoked by the driver xxxThreadInit() routine.

Example 7-2 provides an advanced SCSI controller driver template and Example 7-3 shows a SCSI I/O processor assembly language template. These examples show how such drivers may be structured. Many details are not included in the templates; these templates simply serve to provide a high-level picture of what is involved. Once the basic structure of the template is understood, examining an actual advanced controller driver clarifies the issues involved, especially thread management.

Example 7-2 Advanced Controller Driver Example

/* xxxLib.c - XXX SCSI I/O Processor (SIOP) library */

/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
--------------------
01g,19aug96,dds written */

/*
DESCRIPTION
This is the I/O driver for the XXX SCSI I/O Processor (SIOP). It is designed to work with scsiLib and scsi2Lib. This driver runs in conjunction with a script program for the XXX controller. These scripts use DMA transfers for all data, messages and status. This driver supports cache functions through scsi2Lib.

USER-CALLABLE ROUTINES
Most of the routines in this driver are accessible only through the I/O system. The following routines must be called directly: xxxCtrlCreate() to create a controller structure, and xxxCtrlInit() to initialize it. The XXX SCSI Controller's hardware registers need to be configured according to the hardware implementation. If the default configuration is not proper, the routine xxxSetHwRegister() should be used to properly configure the registers.
This driver supports multiple initiators, disconnect/reconect, tagged command queueing, synchronous data transfer and wide data transfer protocols. In general, the SCSI system and this driver automatically choose the best combination of these features to suit the target devices used. However, the default choices may be over-ridden by using the function "scsiTargetOptionsSet()" (see scsi2Lib).

There are debug variables to trace events in the driver. <scsiDebug> scsiLib debug variable, trace event in scsiLib, xxxScsiPhase(), and xxxTransact(). <scsiIntsDebug> prints interrupt information.

** INCLUDE FILES **

xxx.h, xxxScript.h and scsiLib.h

`/*
#define INCLUDE_SCSI2
#include "vxWorks.h"
#include "memLib.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "logLib.h"
#include "semLib.h"
#include "intLib.h"
#include "errnoLib.h"
#include "cacheLib.h"
#include "taskLib.h"
#include "drv/scsi/xxx.h"
#include "drv/scsi/xxxScript.h"
*/

typedef XXX_SCSI_CTRL SIOP;

/* Configurable options */

int xxxSingleStepSemOptions = SEM_Q_PRIORITY;
char *xxxScsiTaskName = SCSI_DEF_TASK_NAME;
int xxxScsiTaskOptions = SCSI_DEF_TASK_OPTIONS;
int xxxScsiTaskPriority = SCSI_DEF_TASK_PRIORITY;
int xxxScsiTaskStackSize = SCSI_DEF_TASK_STACK_SIZE;

/***********************************************************
 * xxxCtrlCreate - create a control structure for the XXX SCSI controller
 * This routine creates a SCSI controller data structure and must be called
 * before using a SCSI controller chip. It should be called once and only
 * once for a specified SCSI controller. Since it allocates memory
 * for a structure needed by all routines in xodLib, it must be called before
 * any other routines in the library. After calling this routine,
 * xxxCtrlInit() should be called at least once before any SCSI transactions
 * are initiated using the SCSI controller.
*/

206
* RETURNS: A pointer to XXX_SCSI_CTRL structure, or NULL if memory is unavailable or there are invalid parameters.
*
XXX_SCSI_CTRL *xxxCtrlCreate
{
    UINT8 *baseAdrs, /* base address of the SCSI controller */
    UINT   clkPeriod, /* clock controller period (nsec*100) */
    UINT16 devType   /* XXX SCSI device type */
}

{ FAST SIOP *pSiop; /* ptr to SCSI controller info */

/* check that dma buffers are cache-coherent */
/* cacheDmaMalloc the controller structure and other driver structures */
pScsiCtrl = (SCSI_CTRL *) pSiop;
/* inform the SCSI libraries about the size of an XXX event and thread */
pScsiCtrl->eventSize  = sizeof (XXX_EVENT);
pScsiCtrl->threadSize = sizeof (XXX_THREAD);

pScsiCtrl->scsiTransact = (FUNCPTR) scsiTransact;
pScsiCtrl->scsiEventProc = (VOIDFUNCPTR) xxxEvent;
pScsiCtrl->scsiThreadInit = (FUNCPTR) xxxThreadInit;
pScsiCtrl->scsiThreadActivate = (FUNCPTR) xxxThreadActivate;
pScsiCtrl->scsiThreadAbort = (FUNCPTR) xxxThreadAbort;
pScsiCtrl->scsiBusControl = (FUNCPTR) xxxScsiBusControl;
pScsiCtrl->scsiXferParamsQuery = (FUNCPTR) xxxXferParamsQuery;
pScsiCtrl->scsiXferParamsSet = (FUNCPTR) xxxXferParamsSet;
pScsiCtrl->scsiWideXferParamsQuery = (FUNCPTR) xxxWideXferParamsQuery;
pScsiCtrl->scsiWideXferParamsSet = (FUNCPTR) xxxWideXferParamsSet;
/* the following virtual functions are not used with this driver */
pScsiCtrl->scsiDevSelect = NULL;
pScsiCtrl->scsiInfoXfer = NULL;
/* fill in generic SCSI info for this controller */
scsiCtrlInit (&pSiop->scsiCtrl);
/* fill in SCSI controller specific data for this controller */
/* initialize controller state variables */

/* Initialize fixed fields in client shared data area. This "shared" area of memory is shared between this driver and the scripts I/O processor. Fields like data pointers, data size, message pointer, message size, status pointer and size, etc. are typically the pieces of information shared. These fields are updated and managed before and after an I/O process.*/

207
xxxSharedMemInit (pSiop, pSiop->pClientShMem);

/* spawn SCSI manager - use generic code from "scsiLib.c" */

pScsiCtrl->scsiMgrId = taskSpawn (xxxScsiTaskName,
        xxxScsiTaskPriority,
        xxxScsiTaskOptions,
        xxxScsiTaskStackSize,
        (FUNCPTR) scsiMgr,
        (int) pSiop, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

return (pSiop);
}

/***************************************************************************
* xxxCtrlInit - initialize a XXX SCSI controller structure
* This routine initializes an SCSI controller structure, after the structure
* is created with xxxCtrlCreate(). This structure must be initialized before
* the SCSI controller can be used. It may be called more than once if
* needed; however, it should only be called while there is no activity on the
* SCSI interface. A detailed description of the input parameters follows:
* RETURNS: OK, or ERROR if parameters are out of range.
*/

STATUS xxxCtrlInit
{
    FAST XXX_SCSI_CTRL *pSiop, /* ptr to SCSI controller struct */
    int scsiCtrlBusId /* SCSI bus ID of this SCSI controller */

    { 
        SCSI_CTRL * pScsiCtrl = (SCSI_CTRL *) pSiop;

        /* initialize the SCSI controller */
        xxxHwInit (pSiop);

        /* Put the scripts I/O processor in a state whereby it is ready for
        * selections or reselection from the SCSI bus. Such a state continues
        * until either a selection or selection occurs or the driver interrupts
        * the scripts processor and resets its program counter to begin
        * execution elsewhere.
        */

        xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread,
            XXX_SCRIPT_WAIT);

        return (OK);
    }
}
/****************************************************************************
*  xxxHwInit - initialize the SCSI controller chip to a known state
*  RETURNS: N/A
*/
LOCAL void xxxHwInit
{
    FAST SIOP *pSiop /* ptr to a SCSI controller info structure */
}
    /* initialize hardware independent registers */

/****************************************************************************
*  xxxScsiBusReset - assert the RST line on the SCSI bus
*  Issue a SCSI Bus Reset command to the XXX SCSI controller. This should put
*  all devices on the SCSI bus in an initial quiescent state.
*  RETURNS: N/A
*/
LOCAL void xxxScsiBusReset
{
    FAST SIOP *pSiop /* ptr to SCSI controller info */
}
    /* set appropriate register values in order to reset the SCSI bus */

/****************************************************************************
*  xxxIntr - interrupt service routine for the SCSI controller
*  Find the event type corresponding to this interrupt, and carry out any
*  actions which must be done before the SCSI controller is re-started.
*  Determine whether or not the SCSI controller is connected to the bus
*  (depending on the event type - see note below). If not, start a client
*  script if possible or else just make the SCSI controller wait for something
*  else to happen.
*  Notify the SCSI manager of a controller event.
*  RETURNS: N/A
*/
void xxxIntr
{
    SIOP *pSiop
}
    XXX_EVENT event;
    SCSI_EVENT pScsiEvent = (SCSI_EVENT *) &event;
BOOL connected = FALSE;
BOOL notify = TRUE;
int oldState = (int) pSiop->state;

/* Save (partial) SCSI controller register context in current thread */
/* Get event type */
pScsiEvent-type = xxxEventTypeGet (pSiop);
/* fill in event information based upon the nature of the event */
/* controller is now idle: if possible, make it run a script. */
xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread, XXX_SCRIPT_WAIT);
/* Send the event to the SCSI manager to be processed. */
scsiMgrEventNotify ((SCSI_CTRL *) pSiop, pScsiEvent, sizeof (event));
}

/**************************************************************************
* xxxEventTypeGet - parse SCSI and DMA status registers at interrupt time
* RETURNS: an interrupt (event) type code
* /
LOCAL int xxxEventTypeGet
{
   SIOP * pSiop

   /* Read interrupt status registers */
   key = intLock ();
   /* Check for fatal errors first */
   /* No fatal errors; try the rest (order of tests is important) */
   return (INTERRUPT_TYPE);
}

/**************************************************************************
* xxxThreadActivate - activate a SCSI connection for an initiator thread
* Set whatever thread/controller state variables need to be set. Ensure that
* all buffers used by the thread are coherent with the contents of the
* system caches (if any).
* Set transfer parameters for the thread based on what its target device
* last negotiated.
*
* Update the thread context (including shared memory area) and note that *
* there is a new client script to be activated (see "xxxActivate()").
* Set the thread's state to ESTABLISHED.
* Do not wait for the script to be activated. Completion of the script is
* signalled by an event which is handled by "xxxEvent()".
* RETURNS: OK or ERROR
*/
LOCAL STATUS xxxThreadActivate
{
  SIOP * pSiop, /* ptr to controller info */
  XXX_THREAD * pThread /* ptr to thread info */
}
  {
    scsiCacheSynchronize (pScsiThread, SCSI_CACHE_PRE_COMMAND);
    scsiWideXferNegotiate (pScsiCtrl, pScsiTarget, WIDE_XFER_NEW_THREAD);
    scsiSyncXferNegotiate (pScsiCtrl, pScsiTarget, SYNC_XFER_NEW_THREAD);
    if (xxxThreadParamsSet (pThread, pScsiTarget->xferOffset, pScsiTarget->xferPeriod) != OK)
      return (ERROR);
    /* Update thread context; activate the thread */
    xxxThreadUpdate (pThread);
    if (xxxActivate (pSiop, pThread) != OK)
      return (ERROR);
    pScsiCtrl->pThread = pScsiThread;
    xxxThreadStateSet (pThread, SCSI_THREAD_ESTABLISHED);
    return (OK);
  }
}

/***************************************************************************/
* xxxThreadAbort - abort a thread
* If the thread is not currently connected, do nothing and return FALSE to
* indicate that the SCSI manager should abort the thread.
* RETURNS: TRUE if the thread is being aborted by this driver (i.e. it is
* currently active on the controller, else FALSE.
*/
LOCAL BOOL xxxThreadAbort
{
  SIOP * pSiop, /* ptr to controller info */
  XXX_THREAD * pThread /* ptr to thread info */
}
  {
    xxxAbort (pSiop);
    xxxThreadStateSet (pThread, SCSI_THREAD_ABORTING);
return (TRUE);
}

/**************************************************************************
* xxxEvent - XXX SCSI controller event processing routine
* Parse the event type and act accordingly. Controller-level events are
* handled within this function, and the event is then passed to the current
* thread (if any) for thread-level processing.
* RETURNS: N/A
*/
LOCAL void xxxEvent
{
    SIOP * pSiop,
    XXX_EVENT * pEvent
}
{
    SCSI_CTRL * pScsiCtrl = (SCSI_CTRL *) pSiop;
    SCSI_EVENT * pScsiEvent = (SCSI_EVENT *) pEvent;
    XXX_THREAD * pThread = (XXX_THREAD *) pScsiCtrl->pThread;

    /* Do controller-level event processing */
    /* If there's a thread on the controller, forward the event to it */
    if (pThread != 0)
        xxxThreadEvent (pThread, pEvent);
}

/**************************************************************************
* xxxThreadEvent - SCSI controller thread event processing routine
* Forward the event to the proper handler for the thread's current role.
* If the thread is still active, update the thread context (including
* shared memory area) and resume the thread.
* RETURNS: N/A
*/
LOCAL void xxxThreadEvent
{
    XXX_THREAD * pThread,
    XXX_EVENT * pEvent
}
{
    SCSI_EVENT * pScsiEvent = (SCSI_EVENT *) pEvent;
    SCSI_THREAD * pScsiThread = (SCSI_THREAD *) pThread;
    SIOP * pSiop = (SIOP *) pScsiThread->pScsiCtrl;
    XXX_SCRIPT_ENTRY entryPt;

    switch (pScsiThread->role)
    {
    case SCSI_ROLE_INITIATOR:
xxxInitEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_INIT_CONTINUE;
break;

case SCSI_ROLE_IDENT_INIT:
xxxInitIdentEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_INIT_CONTINUE;
break;

case SCSI_ROLE_IDENT_TARG:
xxxTargIdentEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_TGT_DISCONNECT;
break;

case SCSI_ROLE_TARGET:
default:
logMsg (%sxxxThreadEvent: thread 0x%08x: invalid role (%d)\n*, (int) pThread, pScsiThread->role, 0, 0, 0);
entryPt = XXX_SCRIPT_TGT_DISCONNECT;
break;
}

/* Resume thread if it is still connected */
xxxResume (pSiop, pThread, entryPt);
}

/***************************************************************************/
*/
* xxxResume - resume a script corresponding to a suspended thread
* /
* NOTE: the script can only be resumed if the controller is currently idle.
* To avoid races, interrupts must be locked while this is checked and the
* script re-started.
* /
* Reasons why the controller might not be idle include SCSI bus reset and
* unexpected disconnection, both of which might occur in practice. Hence
* this is not considered to be a major software error.
* /
* RETURNS: OK, or ERROR if the controller is in an invalid state (this
* should not be treated as a major software failure).
* /
LOCAL STATUS xxxResume
{
SIOP * pSiop, /* ptr to controller info */
XXX_THREAD * pThread, /* ptr to thread info */
XXX_SCRIPT_ENTRY entryId /* entry point of script to resume */
}
{  
STATUS status;
int key;
/*
 * Check validity of connection and start script if OK
 */
key = intLock();
xxxScriptStart (pSiop, pThread, entryId);
pSiop->state = NCR810_STATE_ACTIVE;
status = OK;
intUnlock (key);
return (status);
}

/***************************************************************************/
*/
* xxxInitEvent - XXX SCSI controller initiator thread event processing route
*/
* Parse the event type and handle it accordingly. This may result in state
* changes for the thread, state variables being updated, etc.
* RETURNS: N/A
*/
LOCAL void xxxInitEvent
(
    XXX_THREAD * pThread,
    XXX_EVENT *  pEvent
)
{
}

/***************************************************************************/
*/
* xxxSharedMemInit - initialize the fields in a shared memory area
*/
* Initialize pointers and counts for all message transfers. These are
* always directed to buffers provided by the SCSI_CTRL structure.
* RETURNS: N/A
*/
LOCAL void xxxSharedMemInit
(
    SIOP * pSiop,
    XXX_SHARED * pShMem
)
{
}

/***************************************************************************/
*/
* xxxThreadInit - initialize a client thread structure
*/
* Initialize the fixed data for a thread (i.e., independent of the command).
* Called once when a thread structure is first created.
*/
/* RETURNS: OK, or ERROR if an error occurs */

LOCAL STATUS xxxThreadInit
{
    SIOP * pSiop,
    XXX_THREAD * pThread
}
{
    scsiThreadInit (&pThread->scsiThread);
    return (OK);
}

/*************************************************************************
* xxxActivate - activate a script corresponding to a new thread
* Request activation of (the script for) a new thread, if possible; do not * wait for the script to complete (or even start) executing. Activation * is requested by signaling the controller, which causes an interrupt. * The script is started by the ISR in response to this event. *
* NOTE: Interrupt locking is required to ensure that the correct action * is taken once the controller state has been checked. *
* RETURNS: OK, or ERROR if the controller is in an invalid state (this * indicates a major software failure). */
LOCAL STATUS xxxActivate
{
    SIOP * pSiop,
    XXX_THREAD * pThread
}
{
    key = intLock ();

    /* Activate controller for the current thread */
    intUnlock (key);

    return (status);
}

/*************************************************************************
* xxxAbort - abort the active script corresponding to the current thread
* Check that there is currently an active script running. If so, set the * SCSI controller Abort flag which halts the script and causes an * interrupt. *
* RETURNS: N/A */
LOCAL void xxxAbort
{
SIOP * pSiop  /* ptr to controller info */
{
  STATUS status;
  int   key;

  key = intLock ();

  /* Abort the active script corresponding to the current thread */
  intUnlock (key);
}

/***************************************************************************/
/* xxxScriptStart - start the SCSI controller executing a script */
/* */
/* Restore the SCSI controller register context, including the shared memory */
/* area, from the thread context.  Put the address of the script entry point */
/* into the DSP register.  If not in single-step mode, start the script. */
/* */
/* NOTE: should always be called with SCSI controller's interrupts locked. */
/* */
/* RETURNS: N/A */
/* */
LOCAL void xxxScriptStart
{
  SIOP *pSiop,  /* pointer to SCSI controller info */
  XXX_THREAD *pThread,  /* ncr thread info */
  XXX_SCRIPT_ENTRY entryId /* routine address entry point */
  
  static ULONG * xxxScriptEntry [] =
  {
    xxxWait,         /* wait for re-select or host cmd */
    xxxInitStart,    /* start an initiator thread */
    xxxInitContinue, /* continue an initiator thread */
    xxxTgtDisconnect,/* disconnect a target thread */
  };

  /* Restore the SCSI controller register context for this thread. */
  /* */
  /* Set the shared data address, load the script start address, */
  /* then start the SCSI controller. */
  */

  /***************************************************************************/
  /* xxxXferParamsQuery - get (synchronous) transfer parameters */
  /* */
  /* Updates the synchronous transfer parameters suggested in the call to match */
  /* the XXX SCSI controller's capabilities.  Transfer period is in SCSI units */
  /* (multiples * of 4 ns). */

  216
LOCAL STATUS xxxXferParamsQuery
{
    SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
    UINT8   *pOffset,    /* max REQ/ACK offset [in/out] */
    UINT8   *pPeriod     /* min transfer period [in/out] */
}
{
    return (OK);
}

 fora status xxxWideXferParamsQuery - get wide data transfer parameters
* 
* Updates the wide data transfer parameters suggested in the call to match
* the XXX SCSI controller's capabilities. Transfer width is in the units
* of the WIDE DATA TRANSFER message's transfer width exponent field. This is
* an 8 bit field where 0 represents a narrow transfer of 8 bits, 1 represents
* a wide transfer of 16 bits and 2 represents a wide transfer of 32 bits.
* 
* RETURNS: OK
* */

LOCAL STATUS xxxWideXferParamsQuery
{
    SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
    UINT8   *xferWidth  /* suggested transfer width */
}
{

}
Example 7-3  Advanced I/O Processor Driver Example

; xxxInit.n Script I/O processor assembly code for xxxLib Driver
;
; Copyright 1989-1996 Wind River Systems, Inc.
;
;Modification history
;---------------------
;01a,28jun95,jds Created. Adapted from ncr710init.n
;
;INTERNAL
;This file contains the assembly level SCSI scripts instructions which are
;used in conjunction with a higher level controller driver. To operate in
;SCSI SCRIPTS mode the SCSI I/O Processor requires only a SCRIPTS start
;address and a signal to begin operation. At that point, the processor
;begins fetching instructions from external memory and then executes them.
;The start address is written to the DMA SCRIPTS Pointer (DSP) register,
;which acts like a typical program counter. All SCRIPT instructions are
;fetched from external memory. The SCSI I/O Processor fetches and executes
;its own instructions by becoming a bus master on the host bus. Instructions
;are executed until a SCSI SCRIPTS interrupt instruction is encountered or
;until an unexpected interrupt causes an interrupt to the external
;processor. Once an interrupt is generated, the SCSI I/O Processor halts all
;operations until the interrupt is serviced. The further execution of
;SCRIPTS is then controlled by the SCSI controller driver which decides
;at which entry point should the SCRIPT processor start executing.
;
;There are four SCRIPT entry points which could be used by the controller
;driver. Execution thereafter is a function of the logic flow within the
SCSI Drivers
7.2 SCSI Overview

;SCRETS and cannot be controlled by the driver. Thus, control is ;transferred to the SCRETS processor by the controller driver at well known ;entry points and this control is returned to the controller driver by the ;SCRETS by generating a SCRETS interrupt. The four SCRETS entry points ;are described below:

1) xxxWait
   If the SCSI controller is not connected to the bus, this entry point is ;used. The SCRETS processor waits for selection or re-selection by a SCSI ;target device (which acts as an initiator during selection), or can be ;interrupted by a new command from the host. This is done by signaling ;the processor via register bits. Thus this entry point puts the SCRETS ;processor into a passive mode.

2) xxxInitStart
   This entry point is used to start a new initiator thread or I/O process ;(in SCSI parlance), selecting a target, sending the identify message and ;thus establishing the ITL nexus, and then continuing to follow the SCSI ;protocol as dictated by the SCSI target, which drives the bus; thus, ;transferring the command, data, messages and status. This processing is ;actually done, within the code of the xxxInitContinue entry point. i.e ;if no stopping condition is encountered, execution continues on into the ;next logical entry point.

3) xxxInitContinue
   This entry point resumes a suspended SCSI thread. SCSI threads are ;when further processing is required by the controller driver and an int ;instruction is executed. However, when the higher level management has ;been worked out, control comes back to a suspended thread and the process ;of cycling through all the SCSI information transfer phases continues. In ;essence, this entry point is the "meat" of an I/O process. The following ;phases are managed by this entry point.
   - DATA_OUT
   - DATA_IN
   - COMMAND
   - STATUS
   - MSG_OUT
   - MSG_IN
   - XXX_ILLEGAL_PHASE

4) xxxTgtDisconnect
   Disconnects a target from the SCSI bus. It is the last entry point in ;an I/O process.

The description level of the code is close to assembly language and is ;in fact the language of the SCRETS processor. The assembly code is compiled ;using an NCR compiler which generates opcodes in the form of a static C ;language structure, which is then compiled and loaded into memory.

The opcode is a pair of 32-bit words, that allow operations and offsets for ;the SCRETS processor. A detailed discussion can be found in the chip's ;programmer's guide. Some of the important instructions and their formats ;are listed below.

block move instruction.
move from <offset> when PHASE_NAME
; I/O instructions
; set target
; wait DISCONNECT
; wait RESELECT
; select from <offset>,@jump
;......
; read/write register instructions
; move REG_NAME to SFBR
; SFBR acts like an accumulator allowing branch instructions based on its
; value
;......
; control transfer instructions
; jump <Label>
; int <value> when PHASE_NAME
;......
; INTERRUPT SOURCES
; The SCSI I/O Processor has three main kinds of interrupt, scsi, dma interrupt
; and script interrupt. The int instruction allows the controller driver to
; be interrupted with an interrupt value which is stored in the DSPS register.
;*/

#define NCR_COMPILE
#include "xxxScript.h"

/**************************************************************************
 */
; xxxWait - wait for re-selection by target, selection by initiator, or
; new command from host
;*/

PROC xxxWait:
; setup instructions here
wait reselect REL(checkNewCmd)
;
; have been re-selected by a SCSI target
reselected:
; handle reselects, insert the reselect logic
int XXX_RESELECTED ; all seems OK so far
;
; may have a new host command to handle
checkNewCmd:
; insert logic for checking if the processor is connected to the bus
int XXX_READY ; processor is ready for a new thread
7 SCSI Drivers
7.2 SCSI Overview

 /**************************************************************************
 /* xxxInitStart - start new initiator thread, selecting target and
 /* continuing to transfer command, data, messages as requested.
 /*
 /* At this point the script requires some data in the scratch registers.
 /* This is the threads context information.
 /*
 /* When the script finishes, these registers are updated with the new context
 /* information
 /*
 */

 PROC xxxInitStart:

 ; If required to identify, select w. ATN and try to transfer IDENTIFY message
 ; (if this fails, continue silently). Otherwise, select without ATN.
 ;
 select atn from OFFSET_DEVICE, REL(checkNewCmd)

 ; add code to test various processor states and conditions interrupt driver
 ; if necessary.

 jump REL(nextPhase)

 /**************************************************************************
 /*
 /* xxxInitContinue - resume an initiator thread
 /*
 /* At this point the script requires the threads context information in
 /* scratch registers
 /*
 /* When the script finishes, these scratch registers are updated with the
 /* the latest context information
 /*
 */

 PROC xxxInitContinue:

 ; some setup code...

 nextPhase:

 ; Normal info transfer request processing
 ;
 phaseSwitch:

 jump REL(doDataOut), when DATA_OUT
 jump REL(doDataIn) if DATA_IN
 jump REL(doCommand) if COMMAND
 jump REL(doStatus) if STATUS
 jump REL(doMsgOut) if MSG_OUT
 jump REL(doMsgIn) if MSG_IN
 int XXX_ILLEGAL_PHASE

 /**************************************************************************
 /*
 /* doDataOut - handle DATA OUT phase
 */
doDataOut:

...  
jump REL(nextPhase)

;**************************************************************************
;*
;* doDataIn - handle DATA IN phase
;*/
doDataIn:

...  
jump REL(nextPhase)

;**************************************************************************
;*
;* doCommand - handle COMMAND phase
;*/
doCommand:

...  
jump REL(nextPhase)

;**************************************************************************
;*
;* doStatus - handle STATUS phase
;*/
doStatus:

...  
jump REL(nextPhase)

;*
;* doMsgOut - handle MSG OUT phase
;*/
doMsgOut:

...  
jump REL(nextPhase)

;**************************************************************************
;*
;* Note: there is little point in having the '810 parse the message type
;* unless it can save the host some work by doing so; DISCONNECT and
;* COMMAND COMPLETE are really the only cases in point. Multi-byte messages
;* are handled specially - see the comments below.
;*/
doMsgIn:
;...
int XXX_MESSAGES_IN_RECVD ; driver handles all others
;
; Have received a DISCONNECT message
; disconn:
;...
int XXX_DISCONNECTED
;
; Have received a COMMAND COMPLETE message
; complete:
;...
int XXX_CMD_COMPLETE
extended:
int XXX_EXT_MESSAGES_SIZE
contExtMsg:
int XXX_MESSAGES_IN_RECVD ; at last!

lcd
 /*********************************************************************/
* xxxTgtDisconnect - disconnect from SCSI bus
* *
*/
PROC xxxTgtDisconnect:
;...
disconnect
int XXX_DISCONNECTED
7.3 SCSI BSP Interface

The BSP provides the board information to the driver in its invocations of the initialization routines. The main tasks of the BSP `sysScsiInit()` routine, which is located in a file named `sysScsi.c` (included from the standard `sysLib.c`), are as follows:

- Address all preliminary board-specific hardware initialization.
- Create a controller driver object by invoking the driver’s `xxxCtrlCreate()` routine and supplying the board-specific hardware information such as the base address to the SCSI controller registers.
- Connect the SCSI controller’s interrupt vector to the driver’s interrupt service routine (ISR).
- Perform additional driver initialization by invoking the `xxxCtrlInit()` routine and optionally the driver’s `xxxHwInit()` routine supplying board-specific information such as the SCSI initiator bus ID, and specific hardware register values.
- Supply any DMA routines if an external DMA controller is being used and is not part of the SCSI controller driver.

Any other board-specific configurations to initialize SCSI peripheral devices such as hard disks and tapes or block/sequential devices and file systems must also be accomplished by `sysScsi.c`. Such configuration initialization shall be located in `sysScsiConfig()`.

The following subsection introduces a template `sysScsiInit()` routine located in `sysScsi.c`.

Example 7-4 Template for SCSI Initialization in the BSP (`sysScsi.c`)

```c
/* sysScsi.c - XXX BSP SCSI-2 initialization for sysLib.c */

/* Copyright 1984-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
 * modification history
 *---------------------
 * 01a,29nov95,jds written
 */

/*
 * Description
 * This file contains the sysScsiInit() and related routines necessary for
```
initializing the SCSI subsystem for this BSP.

/*

#ifdef INCLUDE_SCSI
/* external inclusions */
#include "drv/scsi/xxx.h"
#include "tapeFsLib.h"

/**************************************************************************
* sysScsiInit - initialize XXX SCSI chip
*
* This routine creates and initializes an SIOP structure, enabling use of the
* on-board SCSI port. It also connects the proper interrupt service routine
* to the desired vector, and enables the interrupt at the desired level.
* 
* RETURNS: OK, or ERROR if the control structure is not created or the
* interrupt service routine cannot be connected to the interrupt.
*/

STATUS sysScsiInit ()
{
    /* perform preliminary board specific hardware initializations */
    /* Create the SCSI controller */
    if ((pSysScsiCtrl = (SCSI_CTRL *) xxxCtrlCreate
        {
            (UINT8 *) SCSI_BASE_ADRS,
            (UINT) XXX_40MHZ,
            devType
        }) == NULL)
    {
        return (ERROR);
    }
    /* connect the SCSI controller's interrupt service routine */
    if (intConnect (INUM_TO_IVEC (SCSI_INT_VEC),
        xxxIntr, (int) pSysScsiCtrl) == ERROR)
    {
        return (ERROR);
    }
    /* Enable SCSI interrupts */
    intEnable (SCSI_INT_LVL);
    /* initialize SCSI controller with default parameters (user tuneable) */
    if (xxxCtrlInit ((XXX_SCSI_CTRL *)pSysScsiCtrl,
        SCSI_DEF_CTRL_BUS_ID) == ERROR)
        return (ERROR);
```c
#if (USER_D_CACHE_MODE & CACHE_SNOOP_ENABLE)
    scsiCacheSnoopEnable ((SCSI_CTRL *) pSysScsiCtrl);
#else
    scsiCacheSnoopDisable ((SCSI_CTRL *) pSysScsiCtrl);
#endif
/* Set the appropriate board specific hardware registers for the SIOP */
if (xxxSetHwRegister ((XXX_SCSI_CTRL *)pSysScsiCtrl, &hwRegs)
    == ERROR)
    return(ERROR);
/* Include tape support if configured in config.h */
#endif /* INCLUDE_TAPEFS */
```

### 7.4 The SCSI Driver Development Process

This following are useful tips on how to develop a new SCSI controller. Breaking the project up into small easily managed steps is generally the best approach.

1. Understand the template drivers and the interfaces with the SCSI libraries.
2. Copy the template driver into your new driver directory. Replace the variable routine and macro names with your chosen driver name (for example, `xxxShow()` might become `myDriverShow()`).
3. Make sure that the interrupt mechanism is working correctly so that upon getting a SCSI interrupt, the driver’s ISR is invoked. A good method to ensure that the ISR is invoked is to write to a well known location in memory or NVRAM so that upon re-initialization of the board the developer can tell that the ISR was entered. Getting the ISR to work is a major milestone.
4. Get the driver to select a SCSI peripheral device. A SCSI bus analyzer can clarify what is really happening on the bus, and a `xxxShow()` routine is also
extremely helpful. Selecting a device is the next major milestone.

5. Refine the driver using a standard programming step-wise process until the desired result is achieved.

6. Run the standard Wind River SCSI tests in order to test various aspects of the SCSI bus, including multiple threads, multiple initiators, and multiple peripheral devices working concurrently as well as the performance and throughput of the driver.

7.5 Common SCSI Driver Development Issues

This section discusses common issues and concerns encountered during SCSI driver development. For more information on general driver development issues, see 2.3 Common Design Concerns, p. 8.

7.5.1 Troubleshooting and Debugging

This section provides several suggestions for troubleshooting techniques and debugging shortcuts.

SCSI Cables and Termination

A poor cable connection or poor SCSI termination is one of the most common sources of erratic behavior, of the VxWorks target hanging during SCSI execution, and even of unknown interrupts. The SCSI bus must be terminated at both ends, but make sure that no device in the middle of the daisy chain has pull-up terminator resistors or some other form of termination.

SCSI Library Configuration

Check to see that the test does not exceed the memory constraints within the library, such as the permitted number of SCSI threads, the size of the ring buffers, and the stack size of the SCSI manager. In most cases, the default values are appropriate.
Data Coherency Problems

Data coherency problems usually occur in hardware environments where the CPU supports data caching. First disable the data caches and verify that data corruption is occurring. If the problem disappears with the caches disabled, then the coherency problem is related to caches. (Caches can usually be turned off in the BSP by `#undef USER_D_CACHE_ENABLE`.) In order to further troubleshoot the data cache coherency problem, use `cacheDmaMalloc()` in the driver for all memory allocations. However, if hardware snooping is enabled then the problem may lie elsewhere.

Data Address in Virtual Memory Environments

If the CPU board has a Memory Management Unit (MMU), then you must be careful when setting data address pointers during Direct Memory Access (DMA) transfers. When DMA is used in this environment, the physical memory address must be used instead of the virtual memory address. This is because during DMA transfers from the SCSI bus, the SCSI or DMA controller is the bus master and therefore the MMU on the CPU cannot translate the virtual address to the physical address. Instead, the macro `CACHE_DMA_VIRT_TO_PHYS` must be used when providing the data address to the DMA controller.

7.5.2 Test Suites

The following sections list and describe the tests provided by Wind River. The source code for these test routines is located in the directory `installDir/vxworks-6.x/target/src/test/scsi`.

scsiDiskThruputTest( )

This test partitions a 16MB block device into blocks of sizes 4,096, 65,536, or 1,048,576 bytes. Sectors consist of blocks of 512 bytes. This test writes and reads the block size to the disk drive and calculates the time taken, thus computing the throughput.

Invoke this test as follows:

```
scsiDiskThruputTest "scsiBusId devLun numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

```
scsiBusId
Target device ID
```
7.5 Common SCSI Driver Development Issues

```
deVLun
   Device logical unit ID
numBlocks
   Number of blocks in block device
blkOffset
   Address of first block in volume
```

For example:
```
scsiDiskThruputTest "4 0 0x0000 0x0000"
```

`sctsiDiskTest( )`

This test performs any or all of the tests described below. The invocation for
`sctsiDiskTest( )` is as follows:
```
sctsiDiskTest "test scsiBusId deVLun Iterations numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

- `test`
  One of the following:
  #1: runs only `commonCmdsTest( )`
  #2: runs only `directRwTest( )`
  #3: runs only `directCmdsTest( )`
  -[a]: runs all disk tests

- `scsiBusId`
  Target device ID
- `devLun`
  Device logical unit ID
- `Iterations`
  Number of times to execute read/write tests
- `numBlocks`
  Number of blocks in block device
- `blkOffset`
  Address of first block in volume

For example, the following invocation exercises all disk tests, repeating the
read/write exercise 10 times:
```
scsiDiskTest "-a 4 0 10 0x0000 0x0000"
```
The default test mode is to execute all of the following three tests.

**commonCmdsTest()**  
This test exercises all mandatory SCSI common-access commands for SCSI peripheral devices. These common access commands are:

- TEST UNIT READY
- REQUEST SENSE
- INQUIRY

**directRwTest()**  
This test exercises write, read, and check data pattern for:

- 6-byte SCSI commands
- 10-byte SCSI commands

**directCmdsTest()**  
This test exercises all of the direct-access commands listed below. Optionally, the FORMAT command can be tested by specifying a value of TRUE for the parameter `doFormat`.

- MODE SENSE
- MODE SELECT
- RESERVE
- RELEASE
- READ CAPACITY
- READ
- WRITE
- START STOP UNIT
- FORMAT (optional)

**scsiSpeedTest()**  
This test initializes a block device for use with a dosFs file system. The test uses a large buffer to read and write from and to contiguous files with both buffered and non-buffered I/O.

**scsiSpeedTest()** runs a number of laps, and uses `timex` to time the write and read operations. The speed test should be run on only one drive at a time to obtain maximum throughput.

Invoke this test as follows:

```
scsiSpeedTest "scsiBusId devLun numBlocks blkOffset"
```

The individual parameters must fit the guidelines described below:

- `scsBusId`
  Target device ID

- `devLun`
  Device logical unit ID

- `numBlocks`
  Number of blocks in block device

- `blkOffset`
  Address of first block in volume

For example:

```
scsiSpeedTest "4 0 0x0000 0x0000"
```

### tapeFsTest

This test creates a tape file system and issues various commands to test the tape device. You can choose to test fixed-block-size tape devices, variable-block-size tape devices, or both. Fixed-block tests assume 512-byte blocks.

The invocation for `tapeFsTest` is as follows:

```
tapeFsTest "test scsiBusId devLun"
```

The individual parameters must fit the guidelines described below:

- `test`
  One of the following:
  - `-f` runs only the fixed-block-size test
  - `-v` runs only the variable-block-size test
  - `-a` runs both tests

- `scsBusId`
  Target device ID

- `devLun`
  Device logical unit ID

For example, the following invocation exercises both tests:

```
tapeFsTest "-a 1 0"
```
8

Timestamp Drivers

8.1 Introduction  233
8.2 Timestamp Driver Overview  234
8.3 Timestamp Driver Configuration and BSP Interface  254
8.4 The Timestamp Driver Development Process  258
8.5 Common Timestamp Driver Development Issues  262

8.1 Introduction

Detailed monitoring of real-time application performance requires timing information based on high-resolution timers. You can extend the range of information available from VxWorks kernel instrumentation by supplying a timestamp driver. For example, if a timestamp driver is available, a precise chronology can be displayed by the Wind River System Viewer, a graphical analysis tool for real-time and embedded systems based on VxWorks.

The timer is a hardware facility; a timestamp driver is a software interface to that facility. This document describes the standard interfaces for a VxWorks timestamp driver, and discusses the requirements for a hardware timer to be used with VxWorks kernel instrumentation. It is not a step-by-step tutorial on the process of writing a timestamp driver.
This chapter is meant for the following readers:

- VxWorks users who need to add a timestamp driver to an existing BSP.
- VxWorks users who wish to use an existing VxWorks timestamp driver in their own applications.

This chapter assumes that the reader has a working knowledge of the target board hardware. No knowledge of the VxWorks kernel or of the System Viewer is assumed, although experience writing device drivers is helpful.

### 8.2 Timestamp Driver Overview

This section provides an overview of the timestamp driver environment. It includes information on hardware characteristics as well as information on the VxWorks interface.

#### 8.2.1 Hardware Environment

This section discusses typical hardware timer modes of operation and characteristics. This section also defines the VxWorks requirements for timestamp drivers.

**Modes of Operation**

Most target boards have multiple hardware timers available for operating system and application use. The characteristics of timers vary widely due to evolving hardware technology. However, many different types of timers are suitable for use with VxWorks.

In its most basic form, a timer is simply a timing source (that is, a clock) used as input to a counter. The counter counts up or down as the associated clock transitions.

![Figure 8-1 Basic Form of Timer](basic_form_of_timer.png)
There are three common modes in which timers operate: periodic, one-shot, and timestamp. Many newer timers are versatile and can be used in any one of these modes, depending on how they are configured. The characteristics of each mode are as follows:

Periodic Interrupt Timer
The timer counts up or down to a programmed value (called the terminal count), at which point it generates a hardware interrupt. The counter is reset (either by hardware or software), and begins to count up or down again towards the terminal count. The interrupt is the sole output of a periodic interrupt timer. After acknowledging the interrupt, an interrupt service routine (ISR) usually calls an operating system facility to log the interrupt as a clock tick. In some cases, the ISR calls an application-specific routine instead.

The terminal count may be adjusted so that an interrupt is generated at a specified time interval. For example, if the terminal count is set such that an interrupt occurs every 10 msec, 100 ticks per second are generated (100Hz).

The VxWorks system and auxiliary clocks use the underlying hardware timers in periodic interrupt mode.

One-Shot Timer
The timer counts up or down to a programmed terminal count, at which point it generates a hardware interrupt. The counter is then disabled (either by hardware or software). An ISR acknowledges the interrupt, and then calls a user-specified routine.

Currently, VxWorks does not support a one-shot timer facility in hardware, although this type of timer can be simulated by having a periodic interrupt timer disable the counter in the ISR. One-shot functionality is provided by the watchdog software module.

Timestamp Timer
The timer counts up or down to its maximum count (typically, 0 or MAX_INIT) at which point it generates a hardware interrupt. The counter rolls over and begins to count again towards the maximum value. After acknowledging the interrupt, an ISR calls an operating system facility or application-specific routine to log the counter rollover. At any time, the operating system or application may read the counter value to obtain high-resolution timing information in timestamp tick units.

This mode of operation differs from a periodic interrupt timer in that the counter is usually allowed to count to its maximum value. Additionally, the counter value is the primary output of the timestamp timer, and the interrupt
is only used to announce a counter rollover. Timestamp timer components are typically similar to Figure 8-2.

Figure 8-2 Components of a Timestamp Timer

The remainder of this chapter deals only with timers operating in timestamp mode.

Characteristics of Hardware Timers

Several factors determine how suitable a particular hardware timer may be for a timestamp driver. This information may help you to choose an appropriate timer, if several are available.

Read While Enabled

The most important characteristic of a good timestamp timer is the ability to read the counter’s value without having to stop the timer from counting. If the timer must be disabled to read the timestamp value accurately, the time spent without the timer running is not recorded, although the system is actually doing work and other timers are continuing to run (the system clock, for instance). This situation is commonly called time skew. As time skew accumulates, the timestamp values become more and more removed from the absolute time of the system, as kept by the system clock. Additionally, interrupts must be locked out while the timer is stopped. Both of these effects are detrimental to real-time systems.

Prescaler Counter

The input clock is often passed through a prescaler counter to divide the input clock frequency, thereby producing a lower frequency input for the timestamp counter. Although a prescaler is not always present, it can be a useful way of tuning timer devices that have an unusually high input clock frequency. Using a timer frequency significantly greater than your application demands can hamper real-time performance by increasing the number of cycles spent servicing the timer interrupt.
8 Timestamp Drivers

8.2 Timestamp Driver Overview

Counter Width

The timer’s counter should be at least 16 bits wide, although a 24- or 32-bit counter is preferable. The wider a counter, the less often it must roll over, and therefore the less system overhead its ISR incurs. The input frequency can also be higher with a wide counter, which yields more accurate timing information.

Preload After Disable

Some timers require that the counter be preloaded with a value before counting resumes. This is an issue only for timers that cannot be read while enabled. This characteristic adds to the time spent with the timer disabled, thereby increasing time-skew problems. The preload mechanism itself provides a way to correct skew, but determining the amount of the correction is difficult; see the discussion of counter preloading in 8.4.2 Working Around Deficiencies In Hardware Timers, p.258.

Cache Coherency

As with all hardware devices, the locations of timer device registers must be cache coherent. This ensures that reads and writes to timer registers are actually accessing the register locations themselves, and not CPU data cache locations. If data cache memory exists, and there is no hardware mechanism (such as an MMU) to guarantee data cache coherency for register locations, the timestamp timer driver must make explicit calls to flush and invalidate the CPU’s data cache. This adds to the overhead of reading the timestamp tick value.

VxWorks Requirements for Timestamp Timers

The VxWorks kernel instrumentation uses a timestamp timer, when available, to log timing information for selected operating system events—for example, semaphore gives and takes, task spawns and deletions, system clock ticks, and interrupts.

VxWorks requires that timestamp timers provide the following features:

Rollover Interrupt

The timestamp timer must be able to generate a hardware interrupt once the maximum (or terminal) count is reached. An interrupt is needed to avoid aliasing, by announcing the rollover event. Without the interrupt, timestamps are ambiguous, since there is no way to distinguish two timestamps separated by the timer’s terminal period.
Fine Resolution

The timestamp tick \( \text{resolution} \) is calculated as follows:

\[
\frac{1}{\text{timestamp tick frequency}} = \frac{\text{prescaler}}{\text{input clock frequency}}
\]

To be effective, the resolution should be 10 \( \mu \text{sec} \) or less (that is, a timestamp tick frequency of at least 100 kHz). Although this is not a strict requirement, it is consistent with timing limitations within the VxWorks kernel. If the timestamp timer output is slower than 100 kHz, some instrumented kernel events may not have distinguishable timing information.

Sizable Period

The time between timestamp rollovers is the timestamp timer’s period. The period is defined as the product of the timer resolution and the timer’s maximum count:

\[
\text{period} = (\text{maximum count}) \times \text{resolution}
\]

To be effective, the period should be at least 10 msec. If rollovers are more frequent, the overhead of servicing the rollover interrupt may be too intrusive. The greater the period, the better.

8.2.2 VxWorks OS Interface

This section discusses how your timestamp driver should interface with the VxWorks operating system.

Working with the Wind River System Viewer

Although the timestamp timer is meant to be a general facility, some specific information is needed to use it with the kernel instrumentation support for the System Viewer. This section describes the configuration and attachment of the timestamp driver to the VxWorks kernel instrumentation.
Attaching the Timestamp Driver to VxWorks

Define INCLUDE_TIMESTAMP in installDir/vxworks-6.x/target/config/bspname/config.h to make the timestamp timer routines available to instrumentation logging routines with wvTmrRegister(). This enables the code in usrRoot() (in installDir/vxworks-6.x/target/config/all/usrConfig.c) that connects the timestamp driver to the VxWorks kernel instrumentation package.

If you use the standard routine names (described in 8.3 Timestamp Driver Configuration and BSP Interface, p.254), no other changes are necessary. However, you can also create routines with custom names. This is necessary if a VxWorks timestamp driver is already available for a particular target board, and an alternate driver is to be connected. If this is the case, define INCLUDE_USER_TIMESTAMP as well as INCLUDE_TIMESTAMP (place the definition in installDir/vxworks-6.x/target/config/bspname/config.h), to connect the routines named by the USER_TIMExxx macros instead of the default timestamp routines. This does not change the functionality required for any of the routines. It merely provides the ability to connect routines with different names. The connected routines must still adhere to the requirements and functionality specified in 8.3 Timestamp Driver Configuration and BSP Interface, p.254.

Using the System Clock

The kernel instrumentation expects each rollover event to trigger a call to the timestamp callback routine (saved in the variable sysTimestampRoutine()). As described in section 8.4.3 Using the VxWorks System Clock Timer, p.260, the timestamp driver may use the VxWorks system clock facility. If sysTimestampConnect() returns ERROR, the VxWorks kernel instrumentation assumes the system clock is used, and relies on the system clock tick to signal a timestamp timer rollover event.

Timestamp Driver Components

The component concept has been applied to all timer drivers. Driver components are added to domain and bootable application projects in the same way as any other software component.

The generic TIMESTAMP component is used to describe and define the common API for all timestamp drivers. However, it does not actually add the code to the build system. One timer driver with timestamp capabilities should be added to the

---

1. For more information, see the wvTmrRegister() reference entry.
system build in order to provide the timestamp API entry points. Consult the
documentation on the particular driver to make sure that it provides timestamp
support. Some timer drivers do not provide timestamp support.

Sample Drivers

The following sections contain skeleton code for three different types of timestamp
driver:

- for a hardware timer that can be read while enabled
- for a hardware timer that cannot be read while enabled
- for systems that have no suitable spare timers, thus requiring that timestamps
  be derived from the VxWorks system clock timer

For a description of each of these driver types, see 8.4 The Timestamp Driver
Development Process, p.258. For a template driver that you can use as the basis of
your own timestamp driver, see
installDir/vxworks-6.x/target/src/drv/templateTimer.c.

Example 8-1 Timestamp Drivers for Timers that Can Be Read while Enabled

This example presents a skeleton timestamp device driver for a hardware timer
that can be read while enabled. This type of timer is the simplest to configure for
timestamp mode. See 8.4.1 Timers that Can Be Read While Enabled, p.258, for a
discussion of the most important details involved in writing this kind of driver.

/* sampleATimer.c - sample A timer library */

/* Copyright 1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
modification history
-------------------
01a,23mar94,dzb written.
*/

/*
DESCRIPTION
This library contains sample routines to manipulate the timer functions on
the sample A chip with a board-independent interface. This library handles
the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be
defined.

NOTE: This module provides an example of a VxWorks timestamp timer driver
for a timer that can be read while enabled. It illustrates the structures and routines discussed in the documentation 'Creating a VxWorks Timestamp Driver.' This module is only a template. In its current form, it does not compile.

```c
#ifdef INCLUDE_TIMESTAMP
#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleATimer.h"

/* Locals */
LOCAL BOOL sysTimestampRunning = FALSE; /* running flag */
LOCAL FUNCPTOR sysTimestampRoutine = NULL; /* user rollover routine */
LOCAL int sysTimestampArg = NULL; /* arg to user routine */

/***********************************************************/
* sysTimestampInt - timestamp timer interrupt handler *
* This routine handles the timestamp timer interrupt. A user routine is called, if one was connected by sysTimestampConnect(). *
* RETURNS: N/A *
* SEE ALSO: sysTimestampConnect() *
*/
LOCAL void sysTimestampInt (void)
{
    /* acknowledge the timer rollover interrupt here */
    if (sysTimestampRoutine != NULL) /* call user-connected routine */
        (*sysTimestampRoutine) (sysTimestampArg);
}

/***********************************************************/
* sysTimestampConnect - connect a user rtn to the timestamp timer interrupt *
* This routine specifies the user interrupt routine to be called at each timestamp timer interrupt. It does not enable the timestamp timer itself. *
* RETURNS: OK, or ERROR if sysTimestampInt() interrupt handler is not used. *
*/
STATUS sysTimestampConnect
{
    FUNCPTOR routine, /* routine called at each timestamp timer interrupt */
    int arg /* argument with which to call routine */
}
    {
        sysTimestampRoutine = routine;
        sysTimestampArg = arg;
```
return (OK);
}   
/**************************************************************************
*sysTimestampEnable - initialize and enable the timestamp timer
*This routine connects the timestamp timer interrupt and initializes the
*counter registers. If the timestamp timer is already running, this routine
*merely resets the timer counter. 
*Set the rate of the timestamp timer input clock explicitly within the
*BSP, in the sysHwInit() routine. This routine does not initialize
*the timer clock rate.
* RETURNS: OK, or ERROR if the timestamp timer cannot be enabled.
*/
STATUS sysTimestampEnable (void)
{
    if (sysTimestampRunning)
        {/* clear the timer counter here */
        return (OK);
    }
    /* connect interrupt handler for the timestamp timer */
    (void) intConnect (INUM_TO_IVEC (XXX), sysTimestampInt, NULL);
    sysTimestampRunning = TRUE;
    /* set the timestamp timer’s interrupt vector to XXX (if necessary) */
    /* reset & enable the timestamp timer interrupt */
    /* set the period of timestamp timer (see sysTimestampPeriod()) */
    /* clear the timer counter here */
    /* enable the timestamp timer here */
    return (OK);
}
/**************************************************************************
*sysTimestampDisable - disable the timestamp timer
*This routine disables the timestamp timer. Interrupts are not disabled.
*However, the tick counter will not increment after the timestamp timer
*is disabled, ensuring that interrupts are no longer generated.
* RETURNS: OK, or ERROR if the timestamp timer cannot be disabled.
*/
STATUS sysTimestampDisable (void)
{
if (sysTimestampRunning)
{
  /* disable the timestamp timer here */
  sysTimestampRunning = FALSE;
}

return (OK);

/*****************************************************************************/
* sysTimestampPeriod - get the timestamp timer period
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
* RETURNS: The period of the timer in timestamp ticks.
*/
UINT32 sysTimestampPeriod (void)
{
  /*
   * Return the timestamp timer period here.
   * The highest period (maximum terminal count) should be used so
   * that rollover interrupts are kept to a minimum.
   */
}

/*****************************************************************************/
* sysTimestampFreq - get the timestamp timer clock frequency
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/
UINT32 sysTimestampFreq (void)
{
  UINT32 timerFreq;
  /*
   * Return the timestamp tick output frequency here.
   * This value can be determined from the following equation:
   * timerFreq = clock input frequency / prescaler
   * When possible, read the clock input frequency and prescaler values
   * directly from chip registers.
   */
  return (timerFreq);
struct timespec getTimestampTimers()
{
    int ret;
    ret = sysTimestamp();
    return ret;
}

void setTimestampTimers(struct timespec)
{
    sysTimestampLock();
    // Set the timestamp timers...
    sysTimestampUnlock();
}

void lockUnlockTimestampTimers()
{
    // Lock the timestamp timers...
    // (Use sysTimestampLock() if interrupts are not already locked.)
    // Unlock the timestamp timers...
    // (Use sysTimestampUnlock() if interrupts are already locked.)
}

extern "C" {
    void _start();
}

int main()
{
    // Initialize the timestamp timers...
    // Use sysTimestampInit()...
    //()
    return 0;
}

#endif /* INCLUDE_TIMESTAMP */
Example 8-2  Timestamp Drivers for Deficient Timers

This example presents a skeleton timestamp device driver for a hardware timer that cannot be read while enabled, requires preloading, and counts down. See 8.4.2 Working Around Deficiencies In Hardware Timers, p.258, for a discussion of the most important details involved in writing this kind of driver.

/* sampleBTimer.c - sample B timer library */

/* Copyright 1984-1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
 modification history
 ---------------
 01a,23mar94,dzb written.
 */

/*
 DESCRIPTION
 This library contains sample routines to manipulate the timer functions on the sample B chip with a board-independent interface. This library handles the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be defined.

To support the timestamp timer facility, two timers are used: a counting timer, and a correction timer. The counting timer is used as the timestamp counter, but must be stopped to be read, thereby introducing time skew. The correction timer periodically resets the counting timer in an effort to alleviate cumulative time skew. In addition, the correction timer interrupt is used for one other purpose: to alert the user to a counting timer reset (analogous to a timestamp rollover event).

The TS_CORRECTION_PERIOD macro defines the period of the correction timer, which translates to the period of the counting timer reset (analogous to a timestamp rollover event). The TS_SKEW macro can be used to compensate for time skew incurred when the counting timer is stopped in sysTimestamp() and sysTimestampLock(). The value of TS_SKEW is subtracted from the stopped timestamp counter in an attempt to make up for 'lost' time. The correct value to adjust the timestamp counter is not only board-dependent, it is influenced by CPU speed, cache mode, memory speed, and so on.

NOTE: This module provides an example of a VxWorks timestamp timer driver for a timer that cannot be read while enabled, requires preloading, and counts down. It illustrates the structures and routines discussed in the document "Creating a VxWorks Timestamp Driver." This module is only a template. In its current form, it does not compile.
*/

/* includes */
#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleBTimer.h"
#ifdef INCLUDE_TIMESTAMP
/* defines */
#else
#ifndef TS_CORRECTION_PERIOD
#define TS_CORRECTION_PERIOD 0xXXX... /* timestamp skew correction pd. */
#endif /* TS_CORRECTION_PERIOD */ /* see sysTimestampPeriod() */
#endif

#ifndef TS_SKEW
#define TS_SKEW 0 /* timestamp skew correction time */
#endif /* TS_SKEW */

/* locals */
LOCAL BOOL sysTimestampRunning = FALSE; /* running flag */
LOCAL FUNCPTR sysTimestampRoutine = NULL; /* user rollover routine */
LOCAL int sysTimestampArg = NULL; /* arg to user routine */

/*************************************************************************/
/* sysTimestampInt - correction timer interrupt handler */
/* This routine handles the correction timer interrupt. A user routine is */
/* called, if one was connected by sysTimestampConnect(). */
/* */
/* RETURNS: N/A */
/* */
/* SEE ALSO: sysTimestampConnect() */
/*************************************************************************/
LOCAL void sysTimestampInt (void)
{
    /* acknowledge the correction timer interrupt here */
    sysTimestampEnable ();
    if (sysTimestampRoutine != NULL) /* call user-connected routine */
        (*sysTimestampRoutine) (sysTimestampArg);
}

/*************************************************************************/
/* sysTimestampConnect - connect a user routine to the timestamp timer */
/* This routine specifies the user interrupt routine to be called at each */
/* timestamp timer interrupt. It does not enable the timestamp timer itself. */
/* */
/* RETUNRS: OK, or ERROR if sysTimestampInt() interrupt handler is not used. */
/*************************************************************************/
STATUS sysTimestampConnect
{
    FUNCPTR routine, /* routine called at each timestamp timer interrupt */
    int arg /* argument with which to call routine */

{sysTimestampRoutine = routine;
  sysTimestampArg = arg;
  return (OK);
}

/****************************************************************************
 *
 * sysTimestampEnable - initialize and enable the timestamp timer
 *
 * This routine connects the timestamp timer interrupt and initializes the
 * counter registers. If the timestamp timer is already running, this routine
 * merely resets the timer counter.
 *
 * Set the rate of the timestamp timer input clock explicitly within the
 * BSP, in the sysHwInit() routine. This routine does not initialize
 * the timer clock rate.
 *
 * RETURNS: OK, or ERROR if the timestamp timer cannot be enabled.
 */

STATUS sysTimestampEnable (void)
{
  int lockKey;
  if (sysTimestampRunning)
    {
      lockKey = intLock (); /* LOCK INTERRUPTS */
      /* disable the counting timer here */
      /* preload the reset count here */
      /* enable the counting timer here */
      /* wait for preload to take effect here */
      intUnlock (lockKey); /* UNLOCK INTERRUPTS */
      return (OK);
    }
  /* connect interrupt handler for the correction timer */
  (void) intConnect (INUM_TO_IVEC (XXX), sysTimestampInt, NULL);
  /* set the correction timer’s interrupt vector to XXX (if necessary) */
  sysTimestampRunning = TRUE;
  /* set the period of the correction timer (see sysTimestampPeriod()) */
  /* set the period of the counting timer = reset count */
  /* enable the counting timer here */
/* enable the correction timer here */
/* wait for preload to take effect on both timers here */
return (OK);
}
***************************************************************************
*
* sysTimestampDisable - disable the timestamp timer
*
* This routine disables the timestamp timer. Interrupts are not disabled.
* However, the tick counter will not decrement after the timestamp timer
* is disabled, ensuring that interrupts are no longer generated.
* *
* RETURNS: OK, or ERROR if the timestamp timer cannot be disabled.
* /
STATUS sysTimestampDisable (void)
{
    if (sysTimestampRunning)
    {
        sysTimestampRunning = FALSE;
        /* disable the correction timer here */
        /* disable the counting timer here */
    }
    return (OK);
}
***************************************************************************
*
* sysTimestampPeriod - get the timestamp timer period
*
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
* *
* RETURNS: The period of the timer in timestamp ticks.
* /
UINT32 sysTimestampPeriod (void)
{
    /*
    * Return the correction timer period here.
    * A reasonable correction period should be chosen. A short period
    * causes increased CPU overhead due to correction timer interrupts.
    * A long period allows for a large accumulation of time skew
    * due to sysTimestamp() calls stopping the counting timer.
    */
    return (TS_CORRECTION_PERIOD);
}
8 Timestamp Drivers
8.2 Timestamp Driver Overview

/***************************************************************************
* sysTimestampFreq - get the timestamp timer clock frequency
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/
UINT32 sysTimestampFreq (void)
{
  UINT32 timerFreq;
  /*
   * Return the timestamp tick output frequency here.
   * This value can be determined from the following equation:
   *   timerFreq = clock input frequency / prescaler
   * When possible, read the clock input frequency and prescaler values
   * directly from chip registers.
   */
  return (timerFreq);
}

/***************************************************************************
* sysTimestamp - get the timestamp timer tick count
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
* Call this routine with interrupts locked. If interrupts are
* not already locked, use sysTimestampLock() instead.
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestampLock()
*/
UINT32 sysTimestamp (void)
{
  UINT32 tick = 0;
  register UINT32 * pTick;
  register UINT32 * pPreload;
  if (sysTimestampRunning)
  {
    /* pTick = counter read register location */
    /* pPreload = counter preload register location */
    /* disable counting timer here */
    }
tick = *pTick; /* read counter value */
*pPreload = tick - TS_SKEW; /* set preload value
(with time-skew adjustment) */

/* enable counting timer here */
tick -= (0xfff...); /* adjust to incrementing value */
}

return (tick);
}

/****************************************************************************
 *
 * sysTimestampLock - get the timestamp timer tick count
 *
 * This routine returns the current value of the timestamp timer tick counter.
 * The tick count can be converted to seconds by dividing by the return of
 * sysTimestampFreq().
 *
 * This routine locks interrupts for cases in which it is necessary to stop
 * the tick counter before reading it, or when two independent counters must
 * be read. If interrupts are already locked, use sysTimestamp() instead.
 *
 * RETURNS: The current timestamp timer tick count.
 * SEE ALSO: sysTimestamp()
 */

UINT32 sysTimestampLock (void)
{
    UINT32 tick = 0;
    register UINT32 * pTick;
    register UINT32 * pPreload;
    int lockKey;

    if (sysTimestampRunning)
    {
        lockKey = intLock (); /* LOCK INTERRUPTS */
        /* pTick = counter read register location */
        /* pPreload = counter preload register location */
        /* disable counting timer here */
        tick = *pTick; /* read counter value */
        *pPreload = tick - TS_SKEW; /* set preload value
(with time-skew adjustment) */
        /* enable counting timer here */
        intUnlock (lockKey); /* UNLOCK INTERRUPTS */
        tick -= (0xfff...); /* adjust to incrementing value */
    }

    return (tick);
}
Example 8-3  Timestamp Drivers for the VxWorks System Clock Timer

This example presents a skeleton timestamp driver for systems that have no suitable spare timers, so that timestamps must be derived from the VxWorks system clock timer. See 8.4.3 Using the VxWorks System Clock Timer, p.260, for a discussion of the most important details involved in writing this kind of driver.

/* sampleCTimer.c - sample C timer library */
/* Copyright 1994 Wind River Systems, Inc. */
#include "copyright_wrs.h"
/*
modification history
--------------------
01a,23mar94,dzb written.
*/
/*
DESCRIPTION
This library contains sample routines to manipulate the timer functions on the sample C chip with a board-independent interface. This library handles the timestamp timer facility.

To include the timestamp timer facility, the macro INCLUDE_TIMESTAMP must be defined.

NOTE: This module provides an example of a VxWorks timestamp timer driver implemented by reading the system clock timer counter. It illustrates the structures and routines discussed in the document "Creating a VxWorks Timestamp Driver." This module is only a template. In its current form, it does not compile.
*/
#ifdef INCLUDE_TIMESTAMP
#include "drv/timer/timestampDev.h"
#include "drv/timer/sampleCTimer.h"
/* Locals */
LOCAL BOOL sysTimestampRunning = FALSE; /* running flag */
/**************************************************************************/
* sysTimestampConnect - connect a user routine to the timestamp timer
* interrupt
* This routine specifies the user interrupt routine to be called at each
* timestamp timer interrupt. It does not enable the timestamp timer itself.
*/
#endif /* INCLUDE_TIMESTAMP */
* RETURNS: OK, or ERROR if systimestampInt() interrupt handler is not used. */

STATUS systimestampConnect
{
    FUNCPTR routine, /* routine called at each timestamp timer interrupt */
    int arg /* argument with which to call routine */
}

/* ERROR indicates that the system clock tick specifies a
 * rollover event */
return (ERROR);
}

>Status sysTimestampEnable (void)
{
if (sysTimestampRunning)
    return (OK);

sysTimestampRunning = TRUE;
sysClkEnable (); /* ensure the system clock is running */
return (OK);
}

>Status sysTimestampDisable (void)
{
}
sysTimestampRunning = FALSE;
return (ERROR);
}

/***************************************************************************
* sysTimestampPeriod - get the timestamp timer period
* This routine returns the period of the timer in timestamp ticks.
* The period, or terminal count, is the number of ticks to which the
* timestamp timer counts before rolling over and restarting the counting
* process.
* RETURNS: The period of the timer in timestamp ticks.
*/
UINT32 sysTimestampPeriod (void)
{
    /* return the system clock period in timestamp ticks */
    return (sysTimestampFreq ()/sysClkRateGet ())
}

/***************************************************************************
* sysTimestampFreq - get the timestamp timer clock frequency
* This routine returns the frequency of the timer clock, in ticks per second.
* The rate of the timestamp timer should be set explicitly in the BSP,
* in the sysHwInit() routine.
* RETURNS: The timestamp timer clock frequency, in ticks per second.
*/
UINT32 sysTimestampFreq (void)
{
    UINT32 timerFreq;
    /*
    * Return the timestamp tick output frequency here.
    * This value can be determined from the following equation:
    * timerFreq = clock input frequency / prescaler
    * When possible, read the clock input frequency and prescaler values
    * directly from chip registers.
    */
    return (timerFreq);
}

/***************************************************************************
* sysTimestamp - get the timestamp timer tick count
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
*
* Call this routine with interrupts locked. If interrupts are not already locked, use sysTimestampLock() instead.
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestampLock()
*/

UINT32 sysTimestamp (void)
{
    /* return the system clock timer tick count here */
}
/****************************************************************************
* sysTimestampLock - get the timestamp timer tick count
* This routine returns the current value of the timestamp timer tick counter.
* The tick count can be converted to seconds by dividing by the return of
* sysTimestampFreq().
* This routine locks interrupts for cases in which it is necessary to stop
* the tick counter before reading it, or when two independent counters must
* be read. If interrupts are already locked, use sysTimestamp() instead.
* RETURNS: The current timestamp timer tick count.
* SEE ALSO: sysTimestamp()
*/

UINT32 sysTimestampLock (void)
{
    /* Return the system clock timer tick count here.
     * Interrupts do *not* need to be locked in this routine if
     * the counter does not need to be stopped to be read.
     */
}
#endif /* INCLUDE_TIMESTAMP */

8.3 Timestamp Driver Configuration and BSP Interface

The timestamp timer interface is non-standard; it does not utilize the VxWorks I/O system. Although the interface was developed for use with VxWorks kernel instrumentation, it is also useful as a general BSP facility. The timestamp driver’s external interface may change when a more generic, abstracted timer facility is adopted.
The following sections describe each procedure and its external interface. The descriptions apply to a standard timestamp driver. Although the external functionality must remain as described here, procedure content may differ for a particular driver implementation.

**NOTE:** Remember that each routine must return the appropriate value, as described in the following sections. For example, `sysTimestampEnable()` must return `OK` if successful, or `ERROR` if not successful. (OK and ERROR are defined in the VxWorks header file `installDir/vxworks-6.x/target/h/vxWorks.h`.)

### sysTimestampConnect()

This routine specifies the *timestamp callback routine*, a routine to be run each time the timestamp counter rolls over. If this facility is available, `sysTimestampConnect()` must store the function pointer in the global variable `sysTimestampRoutine` and return `OK`, to indicate success.

If the callback cannot be provided, `sysTimestampConnect()` returns `ERROR` to indicate that no callback routine is connected. In this situation, the VxWorks kernel instrumentation does not use the interrupt handler `sysTimestampInt()` as part of its timestamp timer implementation, but relies instead on the system clock tick to signal a timestamp reset event (see 8.4.3 Using the VxWorks System Clock Timer, p.260). To use the timestamp driver in other applications, you must make similar provisions for an `ERROR` result.

The `sysTimestampConnect()` routine does not enable the timestamp timer itself.

```c
STATUS sysTimestampConnect
(  
  FUNCPTR routine,  
  int  arg  
)
```

The arguments for this routine are the following:

- **routine**  
  Pointer to the routine called at each timer rollover interrupt.

- **arg**  
  Argument for the routine referenced in the *routine* parameter.

The result must be `OK` or `ERROR`. 
**sysTimestampEnable( )**

If the timer is not already enabled, this routine performs all the necessary initialization for the timer (for example, connecting the interrupt vector, resetting registers, configuring for timestamp mode, and so on), and then enables the timestamp timer. If the timer is already enabled, this routine simply resets the timer counter value.

```c
STATUS sysTimestampEnable (void)
```

This routine takes no arguments.

The result must be OK or ERROR.

**sysTimestampDisable( )**

This routine disables the timestamp timer. Interrupts are not disabled. However, the tick counter does not count after the timestamp timer is disabled; thus, rollover interrupts are no longer generated.

```c
STATUS sysTimestampDisable (void)
```

This routine takes no arguments.

The result must be OK or ERROR.

**sysTimestampPeriod( )**

This routine returns the period of the timer in timestamp ticks. The period is the number of ticks the timestamp timer counts before rolling over (or resetting) and restarting the counting process.

```c
UINT32 sysTimestampPeriod (void)
```

This routine takes no arguments.

The result must be the period of the timer in timestamp ticks.

**sysTimestampFreq( )**

This routine returns the output frequency of the timer, in timestamp ticks per second. When possible, the frequency should be derived from actual hardware register values.
If the timer input clock is programmable, do not set its clock rate in `sysTimestampFreq()`. Setting the timer input clock rate should be part of the initialization performed by `sysHwInit()` in `sysLib.c`.

```c
UINT32 sysTimestampFreq (void)
This routine takes no arguments.
The result must be the timestamp timer frequency, in ticks per second.
```

`sysTimestamp()`

This routine returns the current value of the timestamp counter, when interrupts are already locked. To convert this tick count to seconds, divide by the result of `sysTimestampFreq()`. The result must increase; that is, the timestamp values must count up. If you are working with a timer that actually counts down, see `8.4.2 Working Around Deficiencies In Hardware Timers`, p.258.

If interrupts are not already locked, call `sysTimestampLock()` instead.

```c
UINT32 sysTimestamp (void)
This routine takes no arguments.
The result must be the current tick count of the timestamp timer.
```

`sysTimestampLock()`

This routine returns the current value of the timestamp counter. To convert the result to seconds, divide the tick count by the result of `sysTimestampFreq()`. The result must increase monotonically; that is, the timestamp values must count up. If you are working with a timer that actually counts down, see `8.4.2 Working Around Deficiencies In Hardware Timers`, p.258.

This routine locks interrupts for cases in which it is necessary to stop the tick counter in order to read it, or when two independent counters must be read. If interrupts are already locked, call `sysTimestamp()` instead.

```c
UINT32 sysTimestampLock (void)
This routine takes no arguments.
The result must be the current tick count of the timestamp timer.
```
8.4 The Timestamp Driver Development Process

This section discusses the three cases of timestamp device drivers and how each is developed. These descriptions correspond to the sample code provided in Sample Drivers, p.240.

8.4.1 Timers that Can Be Read While Enabled

Example 8-1 shows a sample device driver for hardware timers that can be read while enabled. This type of timer is the simplest to configure for timestamp mode, and the device driver code is straightforward.

Timer Period

The timer should be configured for the highest possible period by setting the terminal count to its maximum value (usually 0xffe... when counting up, and 0 when counting down).

Interrupt Level

If programmable, a high-priority interrupt level should be chosen for boards with a low timer period. This ensures that frequent rollover interrupts are serviced without delay, and that the rollover event is registered in a timely manner with the timestamp callback routine (sysTimestampRoutine()).

Interrupt Locking

Timers that can be read while enabled do not need to lock interrupts in the sysTimestampLock() routine.

8.4.2 Working Around Deficiencies In Hardware Timers

The sample device driver in Example 8-2 illustrates techniques for using a hardware timer that cannot be read while enabled, requires preloading, and counts down. This combination of timer attributes presents several problems for the device driver.
Timer Re-Synchronization

If a timestamp timer cannot be read while enabled, a second correction timer can compensate: use the correction timer to reset the timestamp timer periodically. In this scenario, the second timer runs as a periodic interrupt timer. On each interrupt it resets the first (counting) timer. The counting timer is stopped and read for timestamp values, but never generates an interrupt because it is always reset before reaching its terminal count. However, the correction timer does generate interrupts; because it is not read for timestamp values, it never has time-skew problems. The correction timer ISR resets the counting timer, and then calls the timestamp callback routine (sysTimestampRoutine).

This approach clears the time skew that accumulates in the counting timer between resets. Although a discernible time skew may be present towards the end of the timer period, it is flushed by the reset operation.

Timer Period

Because the counting timer is always reset by the correction timer, the timestamp timer period is really the correction timer period. In the Example 8-2 sample code, this period is set by the TS_CORRECTION_PERIOD macro. The value must balance a short period’s increased interrupt service rate with a long period’s noticeable time skew accumulation.

The chosen period should be based on the amount of time skew that can accumulate, which is related to how often the timestamp facility is called and to the sensitivity of the application using the facility. Wind River’s experience is that a correction period of 100 to 150 msec sufficiently satisfies both requirements for most applications.

Down Counter

The timestamp values must increase. If the timer in use actually counts down, the tick count must be converted to an incrementing value. This is easily done by subtracting the counter value from the reset value (usually 0xfff... for a down counter).
Counter Preloading

If the counter value must be preloaded before the timer can resume counting, three subroutines must perform this action: `sysTimestamp()`, `sysTimestampLock()`, and `sysTimestampEnable()`. The preload operation adds to the time spent with the timer disabled, exacerbating time-skew problems.

After the `sysTimestampEnable()` routine enables the counting timer, it may need to delay until the preload value is physically loaded into the counter. This is an issue for timers that synchronize the preloading with a prescaler output transition. If a delay is not inserted, it may be possible for a fast target board to execute the timer preload, return from `sysTimestampEnable()`, and call `sysTimestamp()`, which stops the timer and specifies a different preload value. This would nullify the `sysTimestampEnable()` reset operation.

Adjustment for Time Skew

Counters that are writeable or that have a preload mechanism can compensate for time skew. While the counter is stopped for a read operation, the counter value or the preload value may be adjusted by adding (for an up counter) or subtracting (for a down counter) the number of ticks spent with the timer disabled. The Example 8-2 sample code subtracts the `TS_SKEW` macro (0, by default) from the stopped timestamp counter in an attempt to make up for lost time. Note that the adjustment value is not only board-dependent, it is influenced by CPU speed, cache mode, memory speed, and so on. In the default case (`TS_SKEW = 0`), compiler optimization eliminates the `TS_SKEW` adjustment.

Counter Read Optimization

Write the `sysTimestamp()` and `sysTimestampLock()` routines so that the counter and preload register locations are set before the timer is stopped, in order to reduce the time spent with the counter disabled. This minor change causes a significant reduction in time skew.

8.4.3 Using the VxWorks System Clock Timer

Example 8-3 presents a sample device driver that reads the VxWorks system clock timer to obtain the timestamp tick count. This approach is useful if there are no
other timers available, and if the system clock timer’s counter can be read while enabled.

**Timer Rollover Interrupt**

When the system clock timer is used as the timestamp timer, the usual `sysTimestampInt()` routine cannot be used to service the timer interrupt. This is because the system clock timer already has an ISR. Thus, the system clock tick can be monitored to provide timestamp rollover information. The `sysTimestampConnect()` routine always returns ERROR because the `sysTimestampRoutine` callback routine is not used.

**Timer Counter Not Reset**

Because the system clock is independent of the timestamp facility, the timestamp driver must not disrupt the system clock in any way. Thus, `sysTimestampEnable()` does not reset the timer counter for the system clock. This causes inaccurate timestamp values until the first system clock tick ISR resets the timer counter. For similar reasons, `sysTimestampDisable()` does not physically disable the system clock.

**Timer Period**

The period of the system clock timer is under the control of the system clock facility, not under the control of the timestamp driver. Thus, the system and the application should not call `sysClkRateSet()` to change the system clock rate once `sysTimestampPeriod()` has been called to determine the timestamp timer period.
8.5 Common Timestamp Driver Development Issues

This sections discusses common issues and concerns encountered during timestamp driver development. For more information on general driver development issues, see 2.3 Common Design Concerns, p.8.

Expect significant changes to the API for all types of timer drivers in the future. Wind River is in the process of developing a new API with an object-oriented interface. This new API corrects the design problem that exists when each driver module provides exactly the same entry points.
9

Additional Drivers

9.1 Introduction

This chapter covers a variety of drivers for different purposes.

9.2 ATAPI Drivers

For most situations, the general purpose ATA/ATAPI driver included with VxWorks (installDir/vxworks-6.x/target/src/drv/hdisk/ataDrv.c) works without modification. The driver uses configurable data access macros which allow the
proper BSP routines to be called when the driver interacts with hardware. ataDrv.c is monolithic, meaning that its routines perform functions that would otherwise be done in a generic library, as well as performing the actual interaction with hardware. Writing a new driver for ATAPI at this time would involve either altering ataDrv.c or extracting its generic functionality, and is not recommended.

9.3 Interrupt Controller Drivers

Interrupt controller drivers work differently in VxWorks 6.x and in VxWorks 5.5. The following sections describe how these drivers are handled for each release.

9.3.1 VxWorks 6.x

For VxWorks 6.x and later, interrupt controllers are incorporated into the processor abstraction layer (PAL), guidelines for writing these drivers are not available at the time of this printing. For more information, see the Wind River Online Support Web site.

9.3.2 VxWorks 5.5

This section briefly discusses creation of interrupt controllers for VxWorks 5.5, which are external to the core processor, whether on-chip or not. Software to deal with interrupt controllers included in the core processor design is usually provided during the architecture port.

In some ways, interrupt controller drivers are among the easiest drivers to write. Unlike most drivers, the external interface to an interrupt controller is almost completely determined by the BSP developer. Neither core OS functionality nor application functionality typically depends on the interrupt controller driver design. A typical interrupt controller driver interface to the BSP is described below.

From another perspective, interrupt controllers are among the more difficult devices to write high quality drivers for. Poor interrupt controller software can have a serious impact on overall system performance. Comments related to interrupts are discussed in several places throughout this document. Before
writing an interrupt controller, all the material related to interrupts should be well understood.

A difficulty in creating interrupt controller drivers is that the architecture support layer provides different functionality for different processor types. For more information, refer to the architecture specific supplement, and to the interrupt related source code in the installDir/vxworks-6.x/target/src/arch/ARCH directory. It is not possible to write a an interrupt controller driver that works with all VxWorks 5.5.x architectures.

Whenever possible, interrupt controller drivers should be written to be configurable with regard to the interrupt controller layout. Even if the interrupt controller is on-chip and the hierarchy is fixed, later revisions of the processor may have additional logical interrupt controllers, and the organization of the interrupt controllers may change. Creating a generic interrupt controller saves time and effort when the processor is upgraded.

Finally, additional difficulty can be avoided in the case of multiple interrupt controller types on a single board, if the overall design follows the comments in the interrupt controller template.

BSP Interface

This section describes a common organization for interrupt controller driver usage, along with some guidelines on specific details of what to avoid and what to make sure is incorporated. Because the design of interrupt controllers varies widely, this can only be an approximate guide. For more information, refer to the template interrupt controller driver in installDir/vxworks-6.x/target/src/drv/intrCtl/templateIntrCtl.c and the interrupt controller driver in your reference BSP.

Typical interrupt controller drivers use two initialization routines. Often, they must provide an interrupt service routine and a connect routine.

Interrupt controllers should be initialized early in sysHwInit(). They must be initialized before any device generates an interrupt. Early in processor initialization, the processor's interrupts are masked, so any interrupts which do occur should not cause problems. Although interrupt controller initialization can occur earlier than this, the best design usually does interrupt controller initialization as the first call from sysHwInit().

Usually, the architecture specific version of intConnect() is called to connect the interrupt controller interrupt source to the architecture specific processor interrupt system. However, intConnect() requires the memory system to be available in
order to allocate memory for a dynamically allocated interrupt stub, which calls
the actual Interrupt Service Routine (ISR).

For this reason, the external interrupt controller cannot be connected to the
architecture specific processor interrupt system until after `sysHwInit()` is
complete and the root task is running. So the appropriate place to put the
`intConnect()` call is at the beginning of `sysHwInit2()`. Usually, the interrupt
controller must be first in `sysHwInit2()`, because it must be before other interrupts
are connected.

**Non-Vectored Interrupt Sources**

In an ideal world, all interrupt sources provide a vector to use for fast interrupt
dispatching. In this case, the hardware provides a vector which is used to dispatch
the appropriate ISR without the need to handle the interrupt controller directly at
interrupt time. Drivers for interrupt controllers with this property may require
nothing more than the `sysHwInit()` and `sysHwInit2()` initialization routines
mentioned above.

However, vectored interrupt sources may not always be available. The interrupt
controller’s output pin is connected to some interrupt input pin on some other
interrupt controller, possibly directly to a processor interrupt, and possibly on
some other interrupt controller elsewhere on the board. Because no vector is
available, the architecture specific interrupt system does not know what device
generated the interrupt. So the interrupt controller driver must query the
controller to see which pin generated the interrupt, and dispatch the appropriate
ISR. How to do this depends on which processor architecture is being used. Refer
to the source code in the `installDir/vxworks-6.x/target/src/arch/ARCH` directory,
the template interrupt controller driver, and any interrupt controller driver in the
reference BSP.

**9.4 Memory Drivers**

Memory controllers are not related to any of the normal device driver interfaces to
the OS. The memory controller is typically configured by boot code early in the
boot process when the processor’s initial power-on initialization is performed, and
not modified afterward.
Memory controllers are often quite simple. The drivers are typically written in assembly and put in the BSP's `romInit.s` file. For example, the assembly source for one PowerPC processor's on-chip memory controller is about 60 lines long, including comments and blank lines, and just 34 assembly instructions when comments and blank lines are removed. Some memory controller initialization sequences are even shorter.

Many times, the assembly source for memory controller initialization is provided by the memory controller vendor. This code can often be used with little or no modification. When additional work is required, it usually takes one of several forms, described in the following sections.

### 9.4.1 Hardware Mismatches

The code provided by the controller vendor may not match the type of memory which is being used in your hardware. In this case, modifications may need to be made to handle bank size, bank count, memory speed and divisors, ECC characteristics, and other aspects of memory configuration. If this is the case, you need to work with the memory controller vendor or memory vendor to determine the appropriate settings. Sometimes, it may be better to re-design the hardware to use memory types that are already supported.

### 9.4.2 Complex Modern Memory Controllers

Memory controllers included on-chip on some modern processors have become complex. In this case, it may be better to write the memory controller in C instead of maintaining hundreds, or even thousands, of lines of assembly source code.

There are a couple of potential problems with this, but solutions may be available. First, in order to use C, a stack must be available for subroutine call overhead. This means some RAM must be available to contain the stack data before the memory controller is configured. If the chip provides a small bank of static RAM, it can sometimes be configured to be available for use by the memory controller driver. If the chip does not already include any on-chip static RAM or if it is not available for other reasons, it may be possible to include a small bank of on-board static RAM for this purpose.

The second problem with using C source code is related to the make subsystem, source code, and the way the bootable image is created. For the purposes of a memory controller driver, the boot ROM image consists of three modules: `romInit.o`, `bootInit.o`, and an object file containing a RAM resident image which is
copied to RAM early in the boot process. A more complete description of the process of creating the boot ROM image or standalone VxWorks image is described in the *VxWorks BSP Developer’s Guide*. Also, you can find details by examining the output of the `make bootrom` command.

The memory controller must be linked into the image along with `romInit.o` and `bootInit.o`, before the RAM resident image is loaded into RAM. The RAM resident image cannot use it, since the RAM must have already been initialized before this image is run.

In both VxWorks 5.5 and VxWorks 6.x, the build system includes a mechanism for including object modules in the RAM-resident image, but it does not include a specific mechanism to include additional object modules in the ROM-resident image.

There is an indirect mechanism to provide object modules which are included in the base bootrom image. Although you cannot include an object module directly, you can include a library in the `LIB_EXTRA` macro in `Makefile`. The full makefile additions might look something like the following, extracted from makefile in the `wrPpmc440gp` BSP:

```makefile
LIB_EXTRA = romExtras.a
# Additional objects used by romInit
EXTRA_OBJS = romI2cDrv.o romSdramInit.o
romExtras.a: $(EXTRA_OBJS)
  $(AR) crus $@ $(EXTRA_OBJS)
```

For additional information, download the `wrPpmc440gp` BSP to see exactly what is being done and how this situation is handled.

### 9.5 Multi-Mode (SIO) Serial Drivers

The generic *multi-mode serial* (SIO) drivers are provided in the `installDir/vxworks-6.x/target/src/drv/sio` directory. These drivers are called SIO drivers to distinguish them from the older serial drivers that have only a single interrupt mode of operation.

SIO drivers provide an interface for setting hardware options, such as the number of stop bits, data bits, parity, and so on. In addition, these drivers provide an
interface for polled communication that can provide external mode debugging (such as ROM-monitor style debugging) over a serial line. Currently only asynchronous-mode SIO drivers are supported.

### 9.5.1 SIO_CHAN and SIO_DRV_FUNCS

Every SIO device is controlled by an SIO_CHAN structure. This structure contains a single member, a pointer to an SIO_DRV_FUNCS structure. These structures are defined in `installDir/vxworks-6.x/target/h/sioLib.h` as:

```c
typedef struct sio_chan /* a serial channel */
{
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
} SIO_CHAN;

typedef struct sio_drv_funcs SIO_DRV_FUNCS;

struct sio_drv_funcs /* driver functions */
{
    int (*ioctl)(
        SIO_CHAN * pSioChan,
        int cmd,
        void * arg
    );

    int (*txStartup)(
        SIO_CHAN * pSioChan
    );

    int (*callbackInstall)(
        SIO_CHAN * pSioChan,
        int callbackType,
        STATUS (*callback)(),
        void * callbackArg
    );

    int (*pollInput)(
        SIO_CHAN * pSioChan,
        char * inChar
    );

    int (*pollOutput)(
        SIO_CHAN * pSioChan,
        char outChar
    );
};
```
The members of the SIO_DRV_FUNCS structure function as follows:

**ioctl**
Points to the standard I/O control interface routine for the driver. This routine provides the primary control interface for any driver. To access the I/O control services for a standard SIO device, use the following symbolic constants:

**txStartup**
Provides a pointer to the routine that the system calls when new data is available for transmission. Typically, this routine is called only from the ttyDrv.o module. This module provides a higher level of functionality that makes a raw serial channel behave with line control and canonical character processing.

**callbackInstall**
Provides the driver with pointers to callback routines that the driver can call asynchronously to handle character puts and gets. The driver is responsible for saving the callback routines and arguments that it receives from the callbackInstall() routine. The available callbacks are SIO_CALLBACK_GET_TX_CHAR and SIO_CALLBACK_PUT_RCV_CHAR.

Define SIO_CALLBACK_GET_TX_CHAR to point to a routine that fetches a new character for output. The driver calls this callback routine with the supplied argument and an additional argument that is the address to receive the new output character if any. The called routine returns OK to indicate that a character was delivered, or ERROR to indicate that no more characters are available.

Define SIO_CALLBACK_PUT_RCV_CHAR to point to a routine the driver can use to send characters upward. For each incoming character, the callback routine is called with the supplied argument, and the new character as a second argument. Drivers normally do not care about the return value from this call. There is usually nothing that the driver could do but to drop a character if the higher level is not able to receive it.

**pollInput** and **pollOutput**
Provide an interface to polled mode operations of the driver. Do not call these routines unless the device has already been placed into polled mode operation by an SIO_MODE_SET operation.

See installDir/vxworks-6.x/target/src/drv/sio/templateSio.c for more information on the internal workings of a typical SIO device driver.
9.5.2 Polled Mode, WDB, and Kernel Initialization

When WDB is used over a serial channel, it puts the SIO driver into polled mode. This mode disables interrupts and performs I/O operations. Eventually, WDB returns the driver to normal interrupt mode operation.

During BSP development, it is possible to use WDB in polled mode before the kernel is available (see the VxWorks BSP Developer's Guide: Porting a BSP to Custom Hardware). In this case, the WDB target agent calls the driver xxxModeSet() routine to set the driver into polled mode. Later, the agent puts the driver back into normal interrupt mode. For more information, see the VxWorks Kernel Programmer's Guide: Kernel.

Your driver must be able to handle this situation. The WDB agent starts the polled mode session by issuing an ioctl() with the SIO_MODE_SET command, which calls the driver xxxModeSet() routine. This routine, as well as the polled mode input and output routines, must be able to function without any previous initialization having been performed.

9.5.3 Serial Ports, WDB, and Interrupts

SIO driver developers must be aware of two issues related to the use of serial ports for a WDB connection in addition to kernel initialization. These issues are related to interrupts and the order of system initialization.

When using a serial port for a WDB connection, WDB switches the port between polled mode and normal operation, depending on what WDB is doing at any given time. During system mode debugging, which is the only debug mode available during system bringup, WDB puts the serial port into polled mode. But at other times, WDB puts the serial port into normal operation, which usually implies an interrupt-driven mode.

Stray interrupts cause the most serious problem. Connecting an interrupt requires that the system memory pool be available. However, during early parts of system initialization, the system memory pool is not yet available. The driver must wait until after usrRoot() begins before it can successfully connect an ISR to the device interrupt. The normal calling sequence is:

usrRoot() => sysClkConnect() => sysHwInit2() => the driver's interrupt initialization routine => intConnect().

If the driver attempts to connect an ISR before usrRoot() runs, the attempt fails. Any subsequent interrupts are stray interrupts, which cause problems during system initialization.
Another problem is related to the behavior of the actual driver if it attempts to connect interrupts before the system has started. In this case, the SIO driver may not function in interrupt mode thereafter, though it should continue to work in polled mode. As mentioned above, interrupts cannot be connected before the system has started.

One possible workaround for both these problems is to write the SIO driver in such a way that it allows the BSP to signal that interrupts cannot be connected. The generic way to do this is to create a global variable in the driver, indicating whether interrupts can be connected. The value should be initialized to TRUE. In the BSP, set the value to FALSE early in sysHwInit(), or in the SYS_HW_INIT_0() macro, if that macro is defined. Then, at the beginning of sysHwInit2(), restore the value to TRUE. Using this mechanism, the driver does not need to be modified to run both on a BSP under development and a standard BSP.

## 9.6 Serial Drivers

**NOTE:** In recent VxWorks releases, multi-mode serial (SIO) drivers have replaced the older serial drivers. This section is provided for developers with legacy code. All new development should use SIO type drivers as described in 9.5 Multi-Mode (SIO) Serial Drivers, p.268.

Generic serial drivers are provided in the `installDir/vxworks-6.x/target/src/drv/serial` directory. Included in this directory is `templateSerial.c` and its corresponding header file, `templateSerial.h`. These files contain detailed information on the design and construction of a typical serial driver. When writing a serial driver, base the driver on this template, then modify the BSP’s `sysLib.c` or `sysSerial.c` files to include the driver as needed.

To manage information about a serial device, `sysLib.c` uses a device descriptor. This device descriptor also encapsulates board-specific information. For example, it typically includes the frequency of the clock and the addresses of the registers, although the details are dictated by the target device. In `sysLib.c`, the serial device descriptor is declared outside the function definitions as:

```c
TY_CO_DEV tyCoDv [NUM_TTY]; /* structure for serial ports */
```

This array is initialized at run-time in `sysHwInit()`. The `TY_CO_DEV` structure is defined in the device header file (for example, `installDir/vxworks-6.x/target/h/`
The following members of the TY_CO_DEV structure are common to all serial drivers:

**tyDev**
- Required by tyLib, the VxWorks tty driver support library.

**created**
- A flag that must be initialized to FALSE in sysHwInit().

**numChannels**
- Used for parameter checking.

The following macros must be defined for all BSPs:

**NUM_TTY**
- Defines the number of serial channels supported. In configAll.h, the default is defined as 2. To override the default, first undefine, then define NUM_TTY in config.h. If there are no serial channels, define NUM_TTY as NONE.

**CONSOLE_TTY**
- This macro defines the channel number of the console. In configAll.h, the default defined as 0. To override the default, first undefine then define CONSOLE_TTY in config.h.
10.1 About VxBus

VxWorks provides the VxBus infrastructure to support device drivers and to define interfaces that device drivers use to interact with the hardware and with the operating system.
There are several reasons for creating the VxBus modules. Vxbus modules:

- Provide a standard API between operating system or middleware functionality and the device support they require.
- Formalize the interface between a device driver and its hardware.
- Make driver support BSP-agnostic from a the perspective of the driver developer and to minimize architecture-specific design decisions in drivers.
- Simplify BSPs by removing bus controller support from the BSP and by removing the requirement that a driver support file be included for every BSP that uses a given driver.
- Allow users to add drivers to VxWorks image projects using the quick-include feature.

**VxBus Hierarchy**

In order to accomplish the goals listed previously, VxBus maintains a database containing information about devices and device drivers. It makes associations between the two sets of information when appropriate and creates an organized way for the operating system and middleware to gain access to device functionality.

VxBus does this by keeping a set of software objects in a hierarchical structure. The form of this hierarchy mimics the hierarchy of the hardware: buses, bus controllers, and devices. In the VxBus model, bus controllers are considered to be devices and therefore have device drivers associated with them.

In addition to the representation of devices and device drivers, there also needs to be a uniform mechanism of manipulating instances. VxBus uses the hierarchical system in order to track what buses are present on the system, what devices are present on each bus, and what devices have been associated with drivers to form instances. Conversely, it tracks what devices have not been associated with a driver and are therefore available to attach to new drivers.

The system starts with a top-level bus, referred to as the processor local bus (PLB). The PLB consists of a bus structure. Each bus structure contains a parent that is a pointer to the instance representing the bus controller for the upstream bus, a list of child devices, and a list of child buses. The parent of the PLB is set to NULL. Early in system initialization, VxBus calls the PLB initialization routine. This routine registers a hard-coded bus element, representing the PLB, with VxBus. As additional bus controllers are discovered by the system, they are registered with VxBus and any downstream buses and devices are configured and initialized.
During system initialization, drivers register with VxBus. After a device and driver are paired together to get an instance, the instance (driver and device) continues through a multi-pass initialization sequence.

10.2 Device Models

VxBus devices can be categorized into three basic models: a register-based device model, a memory device model, and a model to represent processing element devices such as DSPs, FPGAs, and other processors, as well as intelligent devices or devices that are managed with a message-passing interface. Depending on the device design and the bus type on which the device is located, multiple models may be applicable to a single device.

NOTE: In VxWorks 6.2, only the register-based device model is supported.

The three model types are briefly described as follows:

standard register device type
The register device model is used when the device driver configures and manages the device by reading and writing to device registers. This is the standard model used for most device types.

memory device type
The memory model is used for devices that provide RAM to the system. The RAM must be simple RAM and must not have side-effects.

processing element device type
The processing element device model is used for devices that present a more complex interface to the driver. Processing element devices can be complex devices such as DSPs or other processors, or they may be intelligent devices, or they may just be devices that are configured and managed by using a message passing interface such that used by SCSI and USB bus types.
10.3 Terminology

When working with VxBus, proper terminology is important. The introduction of VxBus also introduces new terminology and some existing terminology must be more strictly defined. Key VxBus terms are as follows:

access routine
A routine provided by VxBus that a driver calls in order to access or manipulate a device register.

bus
A hardware mechanism for communication between the processor and a device, or between different devices. In this document, the term bus refers to a specific piece of communication hardware with its associated instance of controller and driver. (See also device, driver, and instance.)

bus controller
The hardware device that controls signals on a bus. The bus controller hardware must be associated with a bus controller device driver in order for VxBus to make use of the device. The service that a bus controller device driver provides is to support the devices downstream from the controller.

bus discovery
See probe.

bus match
A VxBus procedure to create an instance whenever a new device or driver is made available. This procedure is used to determine if a given driver and device should form an instance.

bus type
A kind of bus, such as PCI or RapidIO. See also bus controller.

c

child
A device that is attached to the current bus.

device
A hardware module that performs some specific action, usually visible (in some way) outside the processor or to the external system. See also bus, driver, and instance.

downstream
From the perspective of a device, downstream refers to a point further from the CPU on the bus hierarchy. See also child.
**driver**

A compiled software module, usually consisting of a text segment containing the executable driver code plus a small, static data segment containing information, that is required to recognize whether the driver can manage a particular device. Note that software to configure bus controllers, such as the Raven PCI host controller, is considered a driver. See also *bus, device,* and *instance.*

**driver method**

A specific functionality that may be provided by a driver. Examples of methods include functionality such as suspending a device or reducing power usage. See also *method ID.*

**instance**

A driver and device that are associated with each other. This is the minimal unit that is accessible to higher levels of the operating system. See also *bus, device,* and *driver.*

**method ID**

A *method ID* is the identification of a specific functionality that may be provided by a driver. This must be unique for each method (that is, specific functionality module) on the system. The *method ID* is a number, and is normally expected to be the address of a data variable. See also *driver method.*

**parent**

The bus to which a device is attached.

**probe**

Probe refers to the discovery of devices present on a bus. For some bus types such as PCI, the bus contains information about devices that are present. For those bus types, dynamic discovery is performed during the probe phase. For bus types such as VME, which do not have such functionality, tables are maintained in the BSP that describe the devices that may be present on the system. See also *probe routine.*

**probe routine**

An entry point into drivers. After the system has tentatively identified a device as being associated with a driver, VxBus gives the driver a chance to verify that the driver is suitable to control the device. The driver registers the probe routine to perform this last-minute validation. This routine is optional. If specified, it is normally safe and acceptable for the routine to simply indicate acceptance.
From the perspective of a device, upstream refers to a point closer to the CPU on the bus hierarchy. See also parent.

10.4 Bus Controller Devices

Traditionally, bus controller devices have been considered a special device type. Traditionally, support for bus controller devices is included in the BSP (in VxWorks) or in the operating system (in other operating systems) in an ad-hoc manner. However, with the VxBus infrastructure, bus controller devices are treated like normal devices in most ways. The most significant difference is that regular devices provide a service to the operating system or to middleware and bus controller devices provide a service to VxBus and to other device drivers. In any case, bus controller devices are now treated as devices therefore a device driver is required. For information specific to VxBus bus controller device drivers, see 12. VxBus Bus Controller Device Drivers.

10.5 Services

This section provides an overview of VxBus services. This includes the device discovery process, available interfaces, initialization sequence information, and the bus topology API. For examples of VxBus services, see 13. VxBus Methods and Utilities.

10.5.1 Device Discovery

During startup, VxBus probes for devices present on the system. For each new device, VxBus automatically checks to see whether a driver exists that matches the device. If it finds a match, it attempts to connect the device to the driver and create an instance.
VxBus bus controller device drivers are responsible for determination of what devices are on the bus that is immediately downstream from the bus controller device. At a high level of description, there are three options available: dynamic discovery and configuration (such as is done by PCI); static, table-based discovery and configuration; and external configuration with either dynamic or table-based device discovery. Note that not all bus types have all options available. For more information on these options, see 12.2 Initialization and Registration, p.315.

During post-startup operation, VxBus does not provide a mechanism for discovering insertion of new devices or removal of old devices. However, if (during operation) some module external to VxBus determines that a device insertion event has occurred. It can notify VxBus through the normal API and VxBus will handle the device insertion event in the same manner it handles such events during startup. Device removal events are also handled in a similar manner during operation.

Whenever a new driver module registers with VxBus, either at system startup or during system operation, VxBus checks to see whether there are devices that have not previously been associated with a driver. If it finds any such device, it attempts to link the driver to the device in order to create one or more instances. The user loading the driver is responsible for making sure that the driver's registration routine is called so that VxBus is informed that the driver is available.

10.5.2 VxBus Interfaces

The VxBus provides standard interfaces for access between drivers and devices, between drivers and the OS, and between drivers and VxBus itself. VxBus can also use driver methods to provide new interfaces for a system. The following sections discuss the various interfaces provided by VxBus.

Driver and Device

On creation of an instance, VxBus provides drivers with access to device registers. Because register access can be a significant performance issue in some code sections for certain drivers, the VxBus design provides three levels of optimization: no optimization, internal VxBus optimization, and manual BSP-specific optimization. For more details about the levels of optimization, see 11.4 Device Access in the Register Device Model, p.301.

Driver and OS or Middleware

The mechanism for instances to communicate with the operating system and middleware is through the use of driver methods (see 11.6 Driver Methods, p.310).
The driver registers a set of methods with VxBus and VxBus provides an API for the operating system and middleware to use in order to make calls to instances.

**Driver and VxBus**

VxBus provides an API for drivers to use to interact with VxBus itself and to gain access to services provided by VxBus. This includes connecting ISRs to the appropriate interrupt source as well as finding information about the location of the device in the hardware hierarchy if necessary.

**Defining New Interfaces**

At any time, VxBus can define new interfaces to devices by defining a driver method. Drivers can request services from a driver method from an upstream bus controller driver or from VxBus itself.

### 10.5.3 Initialization Sequence

VxBus defines a three-pass initialization sequence. The first initialization pass occurs before the system is configured. That is, when no operating system facilities are available. The second pass occurs when some operating system facilities are available (for example, memory allocation). The third pass occurs in a separate VxWorks task, asynchronously to system startup. When drivers register with VxBus, they provide an entry point for VxBus to call at each of these initialization stages. For more information on the initialization sequence, see 10.6 Multi-Stage Initialization, p.283.

In general, VxBus APIs are not specific to system startup, but can be called at any time. For example, VxBus is not responsible for detection of device insertion or removal events. However, when an external module detects a device insertion event, it uses the same API to notify VxBus of the event as is used during system startup when a device is detected.

### 10.5.4 Bus Topology

VxBus provides an additional API that may be useful both for device drivers and operating system or middleware modules in order to get information about bus topology relative to an individual device or instance. This API includes the routines `vxbDevParent()` and `vxbDevPath()`.
10 VxBus

10.6 Multi-Stage Initialization

Instances are given several opportunities to perform different types of initialization. This section describes each step of the initialization process as it is done during startup, and briefly discusses some implications of the software environment during each stage.

NOTE: A device driver must not make any assumption about whether the device is in the reset state. The device can be initialized by a boot program, another operating system, or a hardware debugger prior to the time VxWorks is booted. In such a case, the configuration may not be appropriate for the current device driver. In addition, in some cases, it is possible that the probe routine for some other device driver may have modified the registers to a state that makes the device respond in unpredictable ways. Finally, there are some situations where a device configuration can be set to a value that persists across a power cycle. This can happen, for example, when flash devices are set to program mode and then the power is turned off. Power cycling some flash devices does not reset the program mode, therefore the flash device no longer responds to normal flash commands.

vbxDvParent( )
Finds the device ID of the bus controller for the bus on which the current device resides.

VXB_DEVICE_ID vbxDvParent(VXB_DEVICE_ID devID)

vbxDvPath( )
Iterates up through the bus topology, starting at the device and going through each bus in turn, until the PLB is reached. A routine provided as an argument to the call to vbxDvPath() is called at each point in the device tree between the specified device and the PLB.

STATUS vbxDvPath
{
  VXB_DEVICE_ID pBusDv, /* device */
  /* func is the function to call */
  /* at each ctrlr */
  BOOL (*func)(VXB_DEVICE_ID devID, void * pArg),
  void * pArg /* 2nd arg to func */
}
First Stage Initialization

In the first stage of initialization, buses are probed to discover devices present on the bus. This stage of initialization occurs during execution of `sysHwInit()`. Most operating system facilities, including the system memory allocation mechanism, are not available. Bus controller instances should complete their initialization during this stage. Wind River recommends that clocks and timers complete their initialization at this time in order to make themselves available to the operating system, although they cannot connect their interrupts at this time. At a minimum, other instances should set the device to a state in which it does not generate interrupts.

As each device is discovered during the first stage, the bus controller driver announces the discovery to VxBus. At that time, VxBus attempts to match the device with a driver and create an instance. The algorithms used to determine whether a device and driver match each other depend on the bus type. For example, with PCI, the driver must provide the PCI device ID and vendor ID. In addition to the bus-specific test, each driver may optionally provide a probe routine to verify that it can control the hardware. For more information on the probe routine, see 10.6.2 Device Probe Routine, p.285.

Second Stage Initialization

The second stage of initialization occurs during execution of `sysHwInit2()`. At this time, instances have been identified and the first stage of instance initialization has occurred. Virtual memory has been specified therefore drivers can no longer request virtual address ranges for the device registers. The system memory allocation mechanism is available at this time therefore memory can be allocated using the normal means. However, because the console is not yet initialized, it is likely that no console output will be printed even if the `printf()` routine is used.

During the second stage, each instance is given a second chance to initialize itself. Memory can be allocated from the system memory pool for the instance `pDrvCtrl[]` structure or for other use. At this time, the driver can notify VxBus that it wishes to attach ISRs to its interrupts. Wind River recommends that serial devices complete their initialization during this phase because failure to do so precludes them from being used as the system console.
Third Stage Initialization

After all instances have had first-pass and second-pass opportunities to initialize themselves, VxBus gives each instance a chance to perform any final actions in order to prepare the instance for use by higher level operating system modules. For most drivers, connection to ISRs happens at this time. This third stage of initialization is started late in \texttt{usrRoot()}, after the console is initialized but before network initialization is performed. The third stage of initialization is done as a separate task therefore it occurs asynchronously to other parts of operating system initialization.

10.6.1 OS- and Middleware-Specific Initialization

In addition to VxBus initialization, additional initialization can be performed by middleware. In cases where middleware technologies use VxBus facilities to initiate device initialization, the driver installs the driver method during VxBus initialization (for more information, see \textit{11.6 Driver Methods}, p.310). When necessary, driver methods for middleware initialization are expected to be installed during the second stage of VxBus initialization. However, this is not strictly required. Although VxBus provides the mechanism for middleware to initialize devices, VxBus does not initiate any middleware initialization. Instead, the middleware module is responsible for initiating any such initialization. For a description of how to accomplish this kind of initialization, see \textit{VxBus Hierarchy}, p.276.

10.6.2 Device Probe Routine

When the driver registers with VxBus, it provides a set of entry points for initialization. In addition to the entry points for initialization described previously, the driver can optionally provide a routine to probe the device and verify that the driver understands this particular device. This probe routine is used during the formation of an instance. If other criteria are met, if VxBus thinks there is a good match between a device and a driver, and if the driver provides a probe routine, the probe routine is called before creating the instance.

Prior to calling the probe routine, VxBus makes a preliminary assessment that the driver is appropriate to control the device. In fact, the tests done before the probe routine is called are as extensive as is typically done in versions of VxWorks using the legacy driver model. For this reason, a probe routine is not required. However, there are two primary situations where the probe routine is appropriate: when
there are incompatible variants of a particular device and when bus topology has an impact on the device support. These situations are described in the following sections.

- **Incompatible Device Variants**
  
The first situation for using a probe routine is when there are several incompatible variants of a particular device and there are different device drivers available to control each variant. In this case, the standard criteria for matching a driver with a device (such as the device ID and vendor ID for PCI devices) may not be adequate for determining an exact match. A probe routine can perform whatever device-specific actions are required in order to determine the difference between multiple versions, and only accept the versions that it knows it can control. This may include, as an example, reading the PCI sub-device ID and PCI sub-vendor ID in addition to the device ID and vendor ID. Or it may involve checking a status register that has a bit which is reserved (and always zero) in one version but may be non-zero in another version.

- **Topology**
  
The second situation for using a probe routine is when the location of the device in the bus topology has an affect on whether the device is supported in the end product. For example, if a network device has a high-speed bus for use with high-speed network interfaces as well as a low-speed bus to accept a network interface for configuration and diagnostics, placing the low-speed device on the high-speed bus may degrade performance of the high-speed bus enough that the end product fails to meet performance specifications. For example, if the high-speed bus is a PCI-X bus running at 133 MHz and the low-speed bus is a downstream PCI bus running at 33 MHz, plugging a low-speed network interface into the 133 MHz bus may cause the performance of that bus to switch to 33 MHz (due to the low-speed card). In this case, it may not be appropriate to accept the low-speed device if it is placed on the high-speed bus. A probe routine can be written to check the PCI bus number in order to determine whether the device is acceptable at that location and to refuse to manage the device if it is not plugged into the appropriate slot.

When they are necessary, probe routines should be written in such a way as to reduce the risk of problems caused when the device is not the expected device. If your probe routine is poorly written, it can cause a device to be set into a state where it does not respond to normal commands and is therefore not usable by the
device driver that is appropriate to the device. Probe routines should conform to the following guidelines:

- Minimize the amount of interaction with the device before accepting or rejecting it. If the decision can be made based only on information from configuration space, then no regular device registers need to be modified.

- Do not try to fix problems or restore the state of any device unless the driver can determine it is safe to do so. In general, it is better to abandon the device entirely (rather than trying to fix problems) because attempting to restore the device to its default state can further confuse the device status.

A driver should not assume that the device is in its reset state when the driver is attached to it and an instance is created. In the VxWorks device driver legacy model, this is true only because the system may have been rebooted, possibly by another operating system, because the device was powered on or reset. This is still the case. However, although extremely unlikely, it is now possible that another drivers’ probe routine may have modified the device settings.

**NOTE:** Your driver should be aware of the possibility of volatile registers, keyhole registers, device register internal state machines, and other mechanisms that devices use in order to minimize the use of address space. Although the device that the driver is written for may not use any of these mechanisms, a probe routine may be handed a device for which it is not intended and that routine may use these mechanisms. If the device that the driver is written for does use such mechanisms, the driver must assume that the settings are in a non-functional state and reset the mechanism to a usable state.

When modification to regular device registers is required, the driver probe routine may wish to use the register probe routine to manipulate the device register. This avoids problems related to use of `vxMemProbe()` for device registers, such as multiple accesses to a volatile register and transaction size limits.

Furthermore, like `vxMemProbe()`, the register probe routine reads the register after writing it. However, the register probe routine does not fail if the value does not match the value written, while `vxMemProbe()` does. Instead, it succeeds and allows the driver to determine whether the different value indicates success or failure.

The prototype of the routine, if provided, must use a prototype such as the following:

```c
(BOOL *) ({dev}Probe) (VXB_DEVICE_ID devID);
```
10.7 Driver Methods

A driver can provide a list of custom driver methods to inform VxBus and external middleware technologies of services that the driver provides. These driver methods represent services that the driver provides to the operating system and to middleware. Each driver method includes a callable function pointer that performs the service and a method ID to identify which service is performed by that function pointer. These services are activated by use of the `vxbDevMethodRun()` routine (described in `vxbDevMethodRun()`, p. 290), or by a direct call from the middleware after finding the function pointer with `vxbDevMethodGet()`. The method IDs used to represent the driver methods are defined externally to the driver.

Each instance keeps its driver methods in a NULL-terminated table, and the driver installs the methods by modifying the `pMethods` element of the `VXB_DEVICE` structure so that it points to the head of such a table. The table itself can be shared among multiple instances using the same driver. The macro `DEVMETHOD` is available to reference a method ID within the table.

10.7.1 OS and Middleware Access to Driver Methods

VxBus provides an API for other operating system modules to get access to advertised driver methods such as driver unload, power management, and so forth. Another API can be used to perform a given action on all instances, all unattached devices, or both. There are three routines in this API. The usage is as follows:

```c
STATUS vxbDevIterate
(
    (void *)handler(VXBDEVICE_ID devID, void * pArg),
    void * pArg,
    UINT32 flags
)

STATUS vxbDevMethodRun
(
    UINT32 method,
    void * pArg
)

FUNCPTR vxbDevMethodGet
(
    VXBDEVICE_ID devID,
    UINT32 method
)```
When middleware makes a call to `vxbDevIterate()`, the routine provided as the first argument to `vxbDevIterate()` is called once for each appropriate device on the system. A device is considered appropriate for an action if the device is, or is not, associated with a driver to make an instance, depending on the value of the flags variable. Middleware can specify the action be performed on all un-associated devices, on all associated devices, or on both. For example, a fully functional hotswap module may use `vxbDevIterate()` to request that a specific routine be called for all unassigned devices. This routine might check whether a new driver needs to be downloaded and started in order to attach to the device and create an instance.

When middleware makes a call to `vxbDevMethodRun()`, VxBus makes a call to the driver method for each instance supporting the requested functionality (that is, the driver method specified as the first argument). Middleware can specify that a given action be performed by all instances that support that action. For example, a power management module may request that all devices that support power management run the routine that sets power consumption to 50 percent. A filesystem management module may request that all drivers initiate a flush of any disk blocks that might be cached in RAM. In both of these cases, the middleware makes a call to `vxbDevMethodRun()` to have the action performed.

In addition, middleware can have finer-grained control if the middleware is written to know about individual devices. The `vxbDevMethodGet()` routine can be used to query whether a specific instance supports an action, and the routine that is returned can be called manually by the middleware.

The middleware routine that is called from `vxbDevIterate()` and the driver routine that is called from `vxbDevMethodRun()` should have the following prototype:

```c
void (*handler)(VXB_DEVICE_ID devID, void * pArg)
```

Routines that are called by middleware through the `vxbDevMethodGet()` interface must use the same prototype. If this is not the case, the middleware is responsible for insuring that `vxbDevMethodRun()` is never used with that method.

**NOTE:** VxBus defines a mechanism for drivers to provide functionality to operating system modules. However, the format of the data passed between the operating system module and the driver is determined by the operating system module, not by VxBus.
vxbDevIterate( )

This routine runs the specified routine on all devices or instances in the system. The flags field determines whether the specified routine is called for devices (without drivers), for instances (device and driver pairs), or both.

```c
STATUS vxbDevIterate
(FUNCPTR func, /* function to call */
 void * pArg, /* 2nd argument to func */
 UINT32 flags /* specify actions */
)
```

vxbDevMethodGet( )

This routine checks a given device to see whether the device provides support to middleware for the specified driver method. If so, it returns a function pointer to the entry point in the driver for that driver method.

```c
FUNC PTR vxbDevMethodGet
(VXB_DEVICE_ID instance,
 UINT32 method
)
```

vxbDevMethodRun( )

This routine runs the specified driver method routine on each instance supporting the driver method and passes the specified argument to the driver method routine. The format of the argument is determined by the operating system module that defines the driver method.

```c
FUNC PTR vxbDevMethodRun
(UINT32 method,
 void * pArg
)
```

Classes, Types, and Macros

The flags argument for vxbDevIterate() can have any of the following fields set:

VXB_ITERATEINSTANCES
Iterates over all instances. This may be useful, for example, to manipulate devices that are in use or available for use such as for power management.
VXB_ITERATEORPHANS
Iterates over all orphan devices (devices with no matching driver). This may be useful, for example, in determining whether a driver needs to be downloaded in order to make a given device useful.

VXB_ITERATEVERBOSE
Prints diagnostic messages for each device examined and for each device that provides the specified method.

These flags are OR’d together for a call to vxbDevIterate(). This means that it is possible to perform an action on both devices and instances with a single call.

10.8 Register Access

As mentioned previously, devices can be categorized into several classes. For register-based devices, the common register-based device class support is included automatically so that the driver does not need to take any explicit action to gain access to the device registers.

The VXB_DEVICE structure contains an element called pAccess that contains entry points to the routines that drivers must use to manipulate device registers. For portability reasons, VxBus initially installs a set of routines that are designed with portability in mind. These routines allow registers to be read and written regardless of the bus topology leading to the device. However, these portable routines are slow. They may consist of multiple function calls to perform multiple actions. They also have overhead that is associated with keeping track of the current state of the values so that the appropriate transformations (such as byte-order manipulation) can be applied at the appropriate time.

To allow for better performance, VxBus provides a set of optimized routines. Generally, the transformations that might be required are limited to byte-order transformations, selection of addressing space (memory or I/O), and processor pipeline flushes. Routines performing combinations of these common transformations are provided to improve performance. After any given register access using the portable routines, VxBus checks whether the operations that are performed by the portable routines can be done by one of the optimized versions. If so, the optimized version is installed into the pAccess structure of the instance. This eliminates the overhead of multiple function calls as well as the overhead of tracking data format.
10.9 Interrupts

This section defines the VxBus interface used by device drivers to enable and disable interrupts, to connect an ISR to an interrupt, and to manage interrupts during operation.

Within the VxBus framework, a device driver does not need to know the interrupt number or vector that is used to connect the device to the system. Instead, the device driver developer needs to know the number of interrupts that the device can generate. If there is more than one interrupt, the interrupts are treated as indexed elements. The device driver developer needs to know what functionality is appropriate for the interrupt at each index. By convention, in cases where the device generates three interrupts, the order of their functionality in the table is: transmit interrupt, receive interrupt, and error interrupt.

Device driver developers must keep in mind the distinction between interrupts on the device and interrupt routing on the board or processor. When a device generates an interrupt, the device driver typically clears the interrupt source on the device. However, there may be interrupt controllers between the device and the processor and the interrupt condition of each of those interrupt controllers must be cleared in addition to clearing the interrupt source on the device itself. For this reason, VxBus provides a routine to clear and acknowledge interrupts on intervening interrupt controller devices. Each time a device driver clears the interrupt condition on the device, it should call the VxBus interrupt acknowledge routine to handle the intervening interrupt controllers.

Interrupts used by VxBus are generic objects. They are not related to VxWorks kernel interrupt objects. Implementation of VxBus interrupts may be provided at the level of the intConnect() kernel API, or the implementation may skip the intLib layer and work with kernel interrupt objects directly.

10.9.1 VxBus Interrupt Management

The interrupt control functions are global routines called directly (by name) rather than through an entry in the VXB_DEVICE structure. However, the VXB_DEVICE_ID must still be passed to the interrupt routines as a parameter.

The interrupt control routines are as follows (for the routine prototypes, see Interrupt Control Routine API Reference, p.293):

vxIntConnect()

Connects an ISR to the device interrupt. There are restrictions on how the ISR must behave, such as checking a lightweight interlock and checking the device
to see if interrupts were actually generated. For more information, see 11.5 Interrupts, p.308.

vxbIntDisconnect()  
Disconnects the ISR (if any) from the device interrupt.

vxbIntAcknowledge()  
Acknowledges and clears an interrupt from the interrupt controllers, if possible. This routine does not clear the interrupt on the device generating the interrupt.

vxbIntEnable()  
Enables interrupts along the path from the device to the processor. This routine does not enable processor interrupts nor does it enable interrupts on the device itself.

**NOTE:** Processor interrupts are expected to be enabled by the BSP at boot time, and anything that disables interrupts is responsible for re-enabling them. For this reason, drivers can ignore processor interrupts.

vxbIntDisable()  
Disables interrupts along the path from the device to the processor, if possible and if no other devices share the same interrupt path. This routine does not disable interrupts on the device itself. In addition, the ISR for the specified device may be called if the interrupt pin is shared with another device and the other device generates an interrupt. For this reason, use of a lightweight interlock mechanism is recommended.

**Interrupt Control Routine API Reference**

This section provides the API reference information for the VxBus interrupt control routines.

```c
/**************************************************************************
  * vxbIntConnect - connect an ISR to an interrupt
  *
  * This routine connects the specified ISR to the interrupt
  * specified as part of the device handle. ISRs using
  * this mechanism must be able to handle chained interrupt
  * service routines -- they must check whether the device
  * to which they are attached raised an interrupt, instead
  * of simply assuming that the relevant device raised the
  * interrupt.
  *
```
The first ISR connected to a given interrupt line is called directly. However, if additional ISRs are connected to the same interrupt line, then a chain of ISRs is created for this interrupt line. For this reason, each ISR must check whether interrupts have been masked for the device. A convenient way to do this is by using a lightweight interlock mechanism. If interrupts have not been masked, then the ISR must check whether the device actually generated the interrupt.

RETURNS: OK, or ERROR in case of error.

ERRNO: not set

STATUS vxbIntConnect
{
    VXB_DEVICE_ID devID, /* device handle */
    int intNo, /* interrupt number */
    FUNCPTR isr, /* Interrupt Service Routine */
    void * arg /* driver-specific argument */
};

天vxbIntDisconnect - disconnect an ISR from an interrupt

This routine disconnects the specified ISR from the interrupt specified as part of the device handle.

If disconnecting this ISR leaves no ISR attached to the interrupt, then the interrupt line will be disabled if possible. However, device drivers should not assume that it is possible to disable the interrupt, and they should disable interrupt generation on the device before disconnecting their ISR.

RETURNS: OK, or ERROR in case of error.

ERRNO: not set

STATUS vxbIntDisconnect
{
    VXB_DEVICE_ID devID, /* device handle */
    int intNo, /* interrupt number */
    FUNCPTR isr, /* Interrupt Service Routine */
    void * arg /* driver-specific argument */
};

294
/**
 * vxbIntEnable - Enable device interrupts
 *
 * This routine enables the specified interrupt to interrupt the
 * processor. This may involve enabling any number of interrupt
 * controllers between the device and the processor.
 *
 * NOTE: This routine does not enable interrupts on the processor,
 * but only on interrupt controllers in the path between the
 * processor and the device.
 *
 * RETURNS: OK, or ERROR in case of error.
 *
 * ERRNO: not set
 */

STATUS vxbIntEnable
(
    VXB_DEVICE_ID devID,      /* device handle */
    int intNo,               /* interrupt number */
    FUNCPTTR isr,            /* Interrupt Service Routine */
    void  * arg,             /* driver-specific argument */
);  

/**
 * vxbIntDisable - Disable device interrupts
 *
 * This routine disables the specified interrupt to interrupt the
 * processor. This may involve disabling any number of interrupt
 * controllers between the device and the processor.
 *
 * RETURNS: OK, or ERROR in case of error.
 *
 * ERRNO: not set
 */

STATUS vxbIntDisable
(
    VXB_DEVICE_ID devID,      /* device handle */
    int intNo,               /* interrupt number */
    FUNCPTTR isr,            /* Interrupt Service Routine */
    void  * arg,             /* driver-specific argument */
);  

/**
 * vxbIntAcknowledge - clear the status of an interrupt, if possible
 *
 * This routine clears a pending interrupt.
 *
 * RETURNS: OK, or ERROR in case of error.
 *
 * ERRNO: not set
 */
STATUS vxbIntAcknowledge
{
    VXB_DEVICE_ID devID, /* device handle */
    int intNo, /* interrupt number */
    FUNCPTR isr, /* Interrupt Service Routine */
    void * arg /* driver-specific argument */
};
11

VxBus Device Drivers

11.1 Introduction

VxBus-compliant device drivers use the interfaces provided by the VxBus framework to interact with hardware and with the operating system. VxBus device drivers are BSP-agnostic from the perspective of the driver developer which helps to minimize the amount of architecture-specific design elements in drivers.
This chapter describes how a driver uses and interacts with the VxBus framework and also describes what elements are required in a VxBus-compliant device driver.

For more information on the VxBus framework, see 10. VxBus.

11.2 VxBus Registration and Initialization

When each device is discovered, VxBus attempts to use the information in the VXB_DRIVER structure and the information provided by the upstream bus controller to match the device with a driver. There are three levels of tests. The first test is to check whether the busID of the VXB_DRIVER structure matches the busID of the newly discovered device. The second test checks bus-type-specific information by calling the bus type match routine indicated by the bus controller device driver. The third test calls the optional device probe routine (if the driver provides one).

If all three tests succeed, the device and driver are paired up to create an instance. At that time, the initialization of the instance proceeds as described in 11.3 Driver Registration and Initialization, p.298.

11.3 Driver Registration and Initialization

VxBus-compliant drivers must register with the VxBus framework. This is typically done from a driver-specific registration routine that is included in the system during project configuration. This routine does not require arguments. The only required functionality is to make a call to vxbDriverRegister(), and pass a pointer to a structure containing information about the device and driver.
11.3.1 Device Driver Registration

The routine `vxbDriverRegister()` registers a driver with VxBus. Usage is as follows:

```c
void vxbDriverRegister
  (VXB_DRIVER_ID driverID /* per-bus recognition info */)
```

The `VXB_DRIVER` structure is typically included as the first element of a bus-type-specific structure. Additional elements of the bus-type-specific structure are listed below. The structure is typically allocated and initialized at compile time.

The `VXB_DRIVER` fields are:

- **pNext**
  A pointer to the next `VXB_DRIVER` structure. This field can be used to create a linked list of `VXB_DRIVER` structures, one for each bus type that the device understands.

- **devID**
  The type of device. This field contains information that determines whether the device is a bus controller or not.

- **busID**
  The bus type for this `VXB_DRIVER` structure.

- **busVer**
  The VxBus version number. This should be set to the version of VxBus that the driver is created for.

- **drvName**
  The ASCII name of the driver.

- **pDrvBusFuncs**
  A structure containing initialization entry points. There are three entry points, called `devInstanceInit()`, `devInstanceInit2()`, and `devInstanceConnect()`.

- **devProbe**
  A driver probe routine. For more information, see 10.6.2 Device Probe Routine, p.285.

Additional bus-type-specific information typically follows the `VXB_DRIVER` structure.
11.3.2 PLB Registration

The PLB bus type uses the `PLB_DRIVER_REGISTRATION` structure for registration information. The first and only element in this structure is a `VXB_DRIVER` structure.

The PLB cannot be dynamically probed but is always table-driven. The information about PLB devices is kept in the `hcfDeviceList[]` table of the `hwconf.c` file. The PLB reads this table (ignoring any non-PLB devices that are listed) and announces each device to VxBus for initialization. The PLB code sets the `pBusSpecificDevInfo` field of the `VXB_DEVICE` structure to point to the `hcfDeviceList[]` entry.

To check for a driver match, the PLB uses `drvName` and `regBase` to determine whether a device and driver match. If the `drvName` from the `hcfDeviceList[]` entry matches the `drvName` from the `VXB_DRIVER` structure, and if the `regBase` specified in the `hcfDeviceList[]` entry (saved in `pBusSpecificDevInfo`) matches the `regBase[0]` entry in the `VXB_DEVICE` structure, the driver and the device are considered a match.

NOTE: The `drvName` test must match exactly, and is case sensitive. If a driver initialization routine is not being called, the first item you should check is that the `drvName` specified in `hwconf.c` matches the `drvName` specified in the `PLB_DRIVER_REGISTRATION` structure. This can be done easily by calling `vxbShow()` on the running system and checking the driver name and device name.

Because there is no additional information required in order to match the device and driver, the driver does not need to provide additional information in the `PLB_DRIVER_REGISTRATION` structure.

11.3.3 PCI Registration

The PCI bus type uses the `PCI_DRIVER_REGISTRATION` structure for registration information. The first element in this structure is the `VXB_DRIVER` structure. In addition to the generic `VXB_DRIVER` structure, PCI requires a list of PCI device IDs and PCI vendor IDs for each device that the driver can manage. This is provided by a table of `PCI_DEVVEND` structures containing `pciDevID` and `pciVendID` pairs. The `PCI_DRIVER_REGISTRATION` structure contains a pointer to the head of the table (idList), and a count of the number of elements in the table, idListLen.
To check for a driver match, the PCI drive reads the PCI device ID and PCI vendor ID from PCI configuration space, and searches through the `PCI_DEVVEND` table to see if there is a match.

**NOTE:** Some PCI drivers cannot match a device using only the PCI device ID and PCI vendor ID. Drivers for these devices must provide a device probe routine to obtain additional information. The probe routine typically reads additional information from the PCI configuration space, such as subdevice ID and subvendor ID, in order to determine whether the device and driver are a good match. For more information, see *10.6.2 Device Probe Routine*, p.285.

### 11.4 Device Access in the Register Device Model

The register device model is used for device drivers that configure and manage a device by reading and writing to device registers. This section describes how device access is accomplished using this model. (For more information on the register device model, see *10. VxBus*.)

#### 11.4.1 pAccess Structure

For devices that use the register device model, drivers access the device registers by calling one of a number of access routines provided by VxBus when the driver starts. The access routines use the `pAccess` structure pointer from within the `VXB_DEVICE` structure. The access routines are divided into the following categories:

- standard device registers
- registers in configuration space
- volatile registers

In addition, VxBus provides a device register probe routine that is designed to eliminate the problems that occur when `vxMemProbe()` is used to probe device registers.
11.4.2 Standard Device Registers

When working with standard registers, you must keep in mind at all times the tradeoff between increased performance and larger code size (increased footprint). To increase performance, access to device registers should be as fast as possible. This usually requires that code be placed inline at every point that the device register is read from or written to. However, to keep code size small, register access code for complex bus topologies should be put in a routine so that the bulk of the code can be kept in a single location and the many calls to the access routines are simple function calls. Typically, the inline code is used only inside the inner loop of the receive routine and other places where performance is critical. For routines in less performance-critical areas, calls are made to a function to perform the action.

NOTE: In order to maximize performance with minimal footprint, you should use register access macros to access registers during performance critical sections of code where inline code is desirable. Where performance is not critical or where code size is more important than performance, VxBus function pointers should be used instead of macros.

For device drivers, VxBus formalizes this distinction by providing two alternate methods for drivers to access device registers. The normal access to device routines is through routines provided by VxBus. The high-performance access to device routines is done with macros that can be optionally provided by the BSP.

The available register access routines are:

`pAccess->registerRead8()`  
Reads an 8-bit register value. If an 8-bit read cannot be performed, this routine returns an error indication.

`pAccess->registerRead16()`  
Reads a 16-bit register value. If a 16-bit read cannot be performed, this routine returns an error indication.

`pAccess->registerRead32()`  
Reads a 32-bit register value. If a 32-bit read cannot be performed, this routine returns an error indication.

`pAccess->registerRead64()`  
Reads a 64-bit register value. If a 64-bit read cannot be performed, this routine returns an error indication.
11 VxBus Device Drivers

11.4 Device Access in the Register Device Model

pAccess->registerWrite8()

Writes an 8-bit register value. If an 8-bit write cannot be performed, this routine returns an error indication.

pAccess->registerWrite16()

Writes a 16-bit register value. If a 16-bit read cannot be performed, this routine returns an error indication.

pAccess->registerWrite32()

Writes a 32-bit register value. If a 32-bit read cannot be performed, this routine returns an error indication.

pAccess->registerWrite64()

Writes a 64-bit register value. If a 64-bit read cannot be performed, this routine returns an error indication.

Registers are not seen as memory addresses. Instead, registers are accessed by using an identifier, usually a base address, and an offset.

To use the register access routines, you must provide the following:

- a VXB_DEVICE pointer
- the identifier (base address) of the register bank that the driver wants to access
- an offset into that bank of registers
- the data to write, or a pointer to the location to put the newly-read data
- a pointer to a flag for use internal to the VxBus access routines

There are two considerations that require particular attention.

The first consideration is related to the base address parameter. The base address of the register bank must be a copy of one of the pRegBase[ ] entries in the VXB_DEVICE structure. The entry must be copied verbatim and not modified. This is because the VxBus access routines check for validity. This should be considered to be an identifier, rather than a base address. Thus, it is not possible to adjust the base address according to any device-specific criteria. Instead, the offset must be used for that purpose. In addition, the format of the data in the pRegBase[ ] entries may not always contain an address that is valid to the device driver. Instead, the entries may contain an encoded form of the address, such as including the PCI bits to determine in which addressable space the registers belong. Alternatively, the entries may contain an address that is valid for a bus controller downstream, but not for the CPU.

The second consideration is related to the pflags pointer parameter. For normal drivers, this pointer must point to a flag that is isolated to this call, which usually means that the flag must be a local variable. At the beginning of each routine, the flags variable must be set to zero. In general, the driver should not modify the flags
variable. However, if it does modify it, the driver is responsible for restoring the value to zero before any subsequent register accesses.

**Access Routines Provided by VxBus**

During VxBus device driver initialization, a VXB_DEVICE structure pointer is passed to the driver initialization routine. This structure contains an element called pAccess which is a structure containing pointers to the access routines. As described in 10. VxBus, the routines contained in the pAccess structure may install optimized versions of individual routines after the driver is started. This is done by replacing the individual function pointers with pointers to optimized routines. Potentially, this can be done at any time. For this reason, device drivers must not copy the individual routines from the pAccess structure into another location, such as in the pDrvCtrl structure. However, it is safe to copy a pointer to the pAccess structure to another location.

**Optimized Access Routines Provided by the BSP**

The BSP may provide access routines in the form of an optional macro. If defined, the driver can use the macro in performance-critical areas in order to increase performance. Typically, this results in a larger driver code footprint.

The fact that the optimized access routines are provided in the form of a macro has several consequences. First, the device driver must not rely on the presence of optimized access routines. Instead, the driver must check to see whether the macros are available and if not, the driver must use the standard access routines provided by VxBus (see Access Routines Provided by VxBus, p.304). Second, drivers shipped only in source form cannot make use of the BSP-provided routines. These drivers are limited to the standard access routines provided in the pAccess structure element.

The following macros are available:

- `DEV_REGISTER_READ8(devID, regSpace, offset, pDataBuf)`
- `DEV_REGISTER_READ16(devID, regSpace, offset, pDataBuf)`
- `DEV_REGISTER_READ32(devID, regSpace, offset, pDataBuf)`
- `DEV_REGISTER_READ64(devID, regSpace, offset, pDataBuf)`
- `DEV_REGISTER_WRITE8(devID, regSpace, offset, dataBuf)`
- `DEV_REGISTER_WRITE16(devID, regSpace, offset, dataBuf)`
- `DEV_REGISTER_WRITE32(devID, regSpace, offset, dataBuf)`
- `DEV_REGISTER_WRITE64(devID, regSpace, offset, dataBuf)`
BSP-provided optimization is normally only used for very high speed devices such as gigabit Ethernet, some high-performance disk controllers, and some bus controllers themselves (for example, a USB host controller) where high throughput demands inline-speed register access. This is not a requirement, but for most low-speed devices in most applications, the benefits of optimization may not be worth the cost, both in terms of development time and in terms of increased memory footprint for the resulting driver.

**Optimized Access Routines Provided by the BSP and Multiple Bus Locations**

The situation can arise where a single device type resides in multiple bus locations. In such a case, the optimized versions for the access routines cannot be simple branch-free code. Because there is only one copy of the driver loaded on a system, the optimized methods must be able to handle the device regardless of where it is located in the bus topology. For more information about how to create such macros, see the *VxWorks BSP Developer’s Guide*. As a device driver developer, it is sufficient to keep in mind that the optimized register access functionality can be quite large.

**11.4.3 Configuration Space Registers**

Depending on the bus type, some device registers may exist in a separate address space called configuration space. These registers may have a predefined structure such as the PCI configuration space, or they may be device-specific. VxBus directly supports buses that define a single configuration space for each device. Registers in configuration space can be manipulated by use of the VxBus configuration space routines available in the `pAccess` structure. These routines are:

- `pAccess->vxbCfgRead()`
  
  Read from the device configuration space.

- `pAccess->vxbCfgWrite()`
  
  Write to the device configuration space.

The arguments include a `VXB_DEVICE` pointer, the offset into configuration space, the size of the transaction, either the data to write or a pointer to the location to put the newly read data, and a pointer to a flags variable. The flags variable is used in the same manner in configuration routines as it is in standard register routines. For more information on the flags variable, see **11.4.2 Standard Device Registers**, p.302.

VxBus does not put restrictions on the way the operations are performed, such as performing only the specified operation size, but it does require that the
configuration read and write routines conform to the requirements set out by the bus specification. The requirements are as follows:

- **Read a configuration space register of the specified size.**

```c
STATUS (*vxbCfgRead)(
    VXB_DEVICE_ID devID, /* device info */
    UINT32 byteOffset, /* offset into cfg space */
    UINT32 transactionSize, /* transaction size, in bytes */
    void * pDataBuf, /* buffer to read-from/write-to */
    UINT32 * pFlags /* flags */
);
```

- **Write a configuration space register of the specified size.**

```c
STATUS (*vxbCfgWrite)(
    VXB_DEVICE_ID devID, /* device info */
    UINT32 byteOffset, /* offset into cfg space */
    UINT32 transactionSize, /* transaction size, in bytes */
    void * pDataBuf, /* buffer to read-from/write-to */
    UINT32 * pFlags /* flags */
);
```

### 11.4.4 Volatile Registers

Volatile registers present specific problems for device drivers. The volatile register manipulation routines are intended to make it easier for a device driver to manipulate volatile registers.

VxBus provides a mechanism to access volatile device registers. As with standard register access routines and configuration space access routines, the entry points are kept in the `pAccess` structure.

VxBus support for volatile registers does not perform actions such as flushing bus bridge write posting queues. The device driver that uses these routines is responsible for performing such actions.

The entry points are:

- `pAccess->volRegRead()`
  - Read from a volatile device register.

- `pAccess->volRegWrite()`
  - Write to a volatile device register.
The arguments include:

- a `VXB_DEVICE` pointer
- a copy of the `pRegBase[ ]` entry
- the offset into address space
- the size of the transaction
- either the data to write, or a pointer to the location to put the newly read data
- a pointer to a flags variable.

The flags variable use in volatile register routines is the same as that used in standard register routines. For more information on the flags variable, see 11.4.2 Standard Device Registers, p.302.

```c
STATUS (*volRegWrite)(
    VXB_DEVICE_ID devID, /* device info */
    UINT32 regBaseAddr, /* which pRegBase to use */
    UINT32 byteOffset, /* offset, in bytes, of register */
    UINT32 transactionSize, /* transaction size, in bytes */
    void * pDataBuf, /* buffer to read-from/write-to */
    UINT32 * pFlags /* flags */
);
```

The `volRegWrite()` routine writes a value of the specified size to the device register and then reads the value from the register. There is exactly one write transaction and exactly one read transaction, and both transactions are performed with the specified transaction size. The datum written to the device is taken from the variable pointed to by the `pDataBuf` parameter. The value read from the register is written to the variable pointed to by the `pDataBuf` parameter, over-writing the original value.

If the specified transaction size cannot be used, this routine may cause an exception to the calling task or it may return `ERROR`.

```c
STATUS (*volRegRead)(
    VXB_DEVICE_ID devID, /* device info */
    UINT32 regBaseAddr, /* which pRegBase to use */
    UINT32 byteOffset, /* offset, in bytes, of register */
    UINT32 transactionSize, /* transaction size, in bytes */
    void * pDataBuf, /* buffer to read-from/write-to */
    UINT32 * pFlags /* flags */
);
```

The `volRegRead()` routine reads a value of the specified size from the device register. Exactly one transaction is performed, with the specified transaction size.
11.4.5 Register Probe Routine

This register probe routine is similar to \texttt{vxMemProbe()}, but is designed to access device registers instead of memory.

\texttt{pAccess->vxbAddrProbe()}

This routine is used by device drivers to verify that an address is valid and can be read and written with the specified size.

\begin{verbatim}
STATUS (*vxbAddrProbe)
{
    VXB_DEVICE_ID devID, /* device info */
    UINT32 regBaseAddr, /* which pRegBase to use */
    UINT32 byteOffset, /* offset, in bytes, of register */
    UINT32 transactionSize, /* register size */
    UINT32 probeDatum, /* value to write */
    VIRTADDR pRetVal /* value read back */
};
\end{verbatim}

The \texttt{probeDatum} value is written to the device exactly once, and the device register is read exactly once. The value read from the register is returned in the location pointed to by \texttt{pRetVal}.

If a bus error or other access error occurs during the routine, then \texttt{vxbAddrProbe()} returns \texttt{ERROR}, and the value of \texttt{pRetVal} is undefined. This happens when the register does not exist or if the specified transaction size cannot be used to access the register. If no bus error occurs, then \texttt{vxbAddrProbe()} returns \texttt{OK}, and the variable pointed to by \texttt{pRetVal} is set to the value read back from the register.

11.5 Interrupts

VxBus device drivers do not need to know the value of the interrupt pin, line, vector, or IRQ in order to connect the ISR to an interrupt. Instead, VxBus treats interrupts on a per-device basis. For each device, VxBus defines the number of interrupts that are relevant to each device, and which interrupt source corresponds to each area of functionality. This is typically done by the bus type. For example, the standard PCI bus defines four interrupts per device, and each device (or function in PCI terminology) can be connected to one of those four interrupts. For some bus types, different strategies are used.

The VxBus framework requires some information in order to connect an ISR to an interrupt. This includes the interrupt pin, vector, IRQ, or other interrupt
information. For each device, it keeps this information in a table specific to that device. The device driver needs to know only the index of the interrupt in the table in order to connect the ISR. Each instance of a particular device type on a specific bus type has the same organization of interrupt information.

By convention, when three interrupt sources are available on a given device, the interrupts are assigned in the following order:

<table>
<thead>
<tr>
<th>Int</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>transmit interrupt</td>
</tr>
<tr>
<td>1</td>
<td>receive interrupt</td>
</tr>
<tr>
<td>2</td>
<td>error interrupt</td>
</tr>
</tbody>
</table>

Device drivers use the following routines to manipulate interrupts:

- `vxbIntConnect()`
  Connects an ISR to the device interrupt. There are conditions on how the ISR must behave, such as checking a lightweight interlock and checking the device to see if interrupts were actually generated. For more information, see 10.9.1 VxBus Interrupt Management, p.292.

- `vxbIntDisconnect()`
  Disconnects the ISR (if any) from the device interrupt.

- `vxbIntAcknowledge()`
  Acknowledges and clears the interrupt from the interrupt controllers, if possible. This does not clear the interrupt on the device generating the interrupt.

- `vxbIntEnable()`
  Enables interrupts along the path from the device to the processor. This does not enable processor interrupts nor does it enable interrupts on the device itself.

  **NOTE:** Processor interrupts are expected to be enabled by the BSP at boot time and anything that disables interrupts is responsible for re-enabling them. For this reason, drivers can ignore processor interrupts.

- `vxbIntDisable()`
  Disables interrupts along the path from the device to the processor, if possible and if no other devices share the same interrupt path. This does not disable interrupts on the device itself. Also, the ISR for the specified device can be called if the interrupt pin is shared with another device and the other device generates an interrupt. For this reason, use of a lightweight interlock mechanism is recommended.
11.6 Driver Methods

VxBus device drivers advertise some of the services they provide by use of driver methods. A driver method is simply a type of operation that the device is capable of performing. For example, driver de-registration is successful only if the device driver implements a method `VXB_METHOD_DRIVER_UNLINK`. This method is called for every instance using the driver when `vxbDriverUnregister()` is called. This method is also called when `vxbDevRemovalAnnounce()` is called for an instance using this driver.

For more information on driver methods, see 10.7 Driver Methods, p.288. For examples of driver methods and VxBus services, see 13. VxBus Methods and Utilities.

11.7 Multi-Function Devices and VxBus Virtual Buses

When a multi-function devices must be supported, it is a good practice to divide the functionality into different components and provide a separate driver for each component. This allows drivers for unused parts of the device to be scaled out of the system which decreases memory requirements and improves device driver code reusability. However, in certain situations, it is more convenient to manage drivers by including a single unit during configuration of a VxWorks system. Provided the drivers for the individual components of the device are well written, VxBus accommodates this situation easily.

NOTE: In the VxBus paradigm, interrupt pins can be shared among multiple devices. For this reason, drivers must be written to assume that their ISRs may be called even when their interrupts are disabled. For this reason, when an ISR is called, it should verify that interrupts are enabled using, for example, a lightweight interlock, and it should verify that the device has actually generated an interrupt.
In general, top-level drivers for multi-function devices should not manage the hardware at all. Management of the hardware should be left to the drivers for each individual device function. Although there may be situations where it seems logical for a multi-function driver to manage certain aspects of the device (such as registers that are shared among different parts of the device), well designed drivers for the constituent device functions should eliminate this need.

Instead, the multi-function driver should simply manage the individual function drivers. In a sense, the top-level multi-function driver can be considered to be a trivial bus controller that manages the bus internal to the device, though without the overhead of traditional bus support. The name for multi-function drivers should always begin with the letters `mf`, and that the subsequent letter should be capitalized according to the normal Wind River naming convention. For example, the Motorola CPM device that is included in many PowerPC processors, might have a multi-function device named `mfMotCpm`. When the system discovers that `mfMotCpm` is available on the system, it initializes the `mfMotCpm` driver using the normal VxBus initialization sequence. However, instead of initializing the hardware, the `mfMotCpm` initialization routine simply announces to VxBus (using the normal VxBus API) that an interrupt controller, serial devices, network devices, and other constituent functions are available.

### 11.8 Common Issues

You may encounter one or more of the following issues while developing your VxBus device driver.

- **Matching the Configured Device Name in PLB Drivers**
  
  PLB drivers must match the name with the configured device name exactly.

- **Calling the vxbDevRegInfo::devProbe Structure Multiple Times**
  
  The `vxbDevRegInfo::devProbe` structure can be called multiple times if there are multiple devices on the underlying bus. This must be handled when writing the device probe routine.

- **Using the vxbDevRegInfo::pNext Structure During De-Registration**
  
  The `vxbDevRegInfo::pNext` structure can be a valid pointer during driver registration, but it cannot be used during driver de-registration.
### Calling `vxbDeviceAnnounce()` When VXB_DEVICE_ID::pParentBus is Invalid

Before calling `vxbDeviceAnnounce()` `VXB_DEVICE_ID::pParentBus` must be a valid pointer. If it is not valid, the device is added to the lost devices list and then becomes inaccessible.

### NULL Function Pointers in the `vxbDevRegInfo->pDrvBusFuncs` Structure

Function pointers in the `vxbDevRegInfo->pDrvBusFuncs` structure must not be `NULL`. If a function pointer is `NULL`, the system will crash.

### Defining Method IDs as Constants

Method IDs need to be defined as addresses rather than constants. This avoids contention among method IDs defined in multiple drivers.

## 11.9 Debugging Your Device Driver

The VxBus framework is designed to assist you when debugging your device driver development code. This includes the use of VxBus show routines and VxBus debug messages. This section discusses these built-in debug methods.

### 11.9.1 VxBus Show Routines

The `vxBusShow()` routine can be used to show details of the complete VxBus subsystem such as:

- a list of device drivers and bus types that are registered
- a list of buses and the orphans or instances associated with each bus

In addition, the `vxbBusListPrint()` routine can be used to print information about the bus hierarchy.

The timer library provides a `vxbTimerShow()` routine that can be used to list the timers available in the system.

The `vxbTopoShow()`, `vxbDevStructShow()`, `vxbDevAccessShow()`, and `vxbDevPathShow()` can also be used to aid in the debugging process.

For more information on these routines, see the corresponding reference entries.
11.9.2 VxBus Debug Messages

When the \texttt{VXB_DEBUG\_MSG} macro is defined, it can be used to get debug messages from VxBus.

In addition, the macro \texttt{VXB\_TIMER\_DEBUG} can be defined to get messages from the VxBus timer abstraction layer. For more information, see 13.3 Timer Abstraction Routines, p.321.

11.10 Integrating Your Driver Code with VxBus

When integrating your driver with VxBus, your files should be placed in an appropriate directory under $\text{installDir}/vxworks-6.x/target/src/hwif$. If your newly developed files provide features that are used by middleware technologies, the header files should be placed in an appropriate directory under $\text{installDir}/vxworks-6.x/target/h/hwif$.

Once the driver is ready for testing, the following changes should be made depending on the requirements for building the source code:

- Building using a Command Line BSP build
  
  If you are building from the command line, appropriate changes should be made to $\text{installDir}/vxworks-6.x/target/src/hwif/util/cmdLineBuild.c$.

- Building from a Project
  
  If you are building from a project, appropriate changes should be made to $\text{installDir}/vxworks-6.x/target/config/comps/vxWorks/10vxBus.cdf$.

\begin{tabular}{|l|}
\hline
\textbf{NOTE:} Before building your source code, your BSP should provide VxBus support. For more information, see the VxWorks BSP Developer’s Guide. \\
\hline
\end{tabular}
12 VxBus Bus Controller Device Drivers

12.1 Introduction

In the VxBus driver model, bus controller drivers are considered to be standard device drivers and are expected to conform to the same system as other standard device drivers. Bus controller device drivers use VxBus facilities to gain access to the bus controller hardware, and they provide VxBus services to downstream devices. If you are developing a VxBus bus controller device driver, you should be familiar with standard VxBus device drivers (for more information, see II. VxBus Device Drivers).

12.2 Initialization and Registration

The first-pass driver initialization routine (as described in First Stage Initialization, p. 284) is intended primarily for bus controller devices. Bus controller devices must allocate a DRV_CTRL structure using the hwMemAlloc() routine. The bus
controller must be initialized, and the bus must be announced to VxBus with a call to \texttt{vxbBusAnnounce()}. The bus controller device driver is responsible for device enumeration. Depending on the system configuration options, one of three versions may be present: dynamic discovery and configuration, table-based static discovery and configuration, and external configuration. However, as devices are discovered, each new device must be announced to VxBus with a call to \texttt{vxbDeviceAnnounce()}. The following routines are provided by VxBus for registration of devices and bus types.

\texttt{vxbBusTypeRegister()}  
Registers a bus type (for example, PCI) with VxBus.

\begin{verbatim}
void vxbBusTypeRegister

{  VXB_BUSTYPE_ID pBusType
}
\end{verbatim}

\texttt{vxbBusAnnounce()}  
Creates a new structure to represent an example of the specified bus type. A device driver representing a bus controller calls this routine to announce to VxBus that it is a bus controller and that there is a bus downstream from the controller.

\begin{verbatim}
STATUS vxbBusAnnounce

{  VXB_DEVICE_ID pBusDev, /* bus controller */
   UINT32 busID /* bus type */
}
\end{verbatim}

\texttt{vxbDeviceAnnounce()}  
Announces that a new device has been discovered. Bus controller device drivers must call this routine when they discover additional devices. If a driver matches the device, an instance is created. If no driver matches the device, VxBus keeps information about the device in case a driver is later downloaded.

\begin{verbatim}
STATUS vxbDeviceAnnounce

{  VXB_DEVICE_ID pBusDev
}
\end{verbatim}
13.1 Introduction

This chapter provides examples of the methods and utilities (services) provided by VxBus including the VxBus parameter system and the timer abstraction layer.

For more information on VxBus services, see 10.5 Services, p.280.

13.2 VxBus Parameter System

Device parameters are values that are useful for the proper functioning of a device. The device parameters are normally identified by a name and represented as an ASCII string. The VxBus parameter system:
Enables device drivers to set the default values for device parameters.

- Allows BSPs to override the default device parameter values.
- Allows VxWorks middleware technologies to override the parameter values specified by the device driver and the BSP overrides.

The VxBus parameter system also provides interfaces for obtaining the current parameter values.

13.2.1 VxBus Parameter System Functions

The device driver, during its registration with VxBus, provides a table with the default values for the device parameters. Each entry of the table should include:

- a parameter name
- a parameter type
- a parameter value

If the device driver registers the parameter table and an instance is created, the parameters take the default values from the parameter table.

The BSP can override the default parameter values by providing its own table, where each entry includes the following information:

- an instance name
- a unit number
- a parameter name
- a parameter type
- a parameter value

The middleware can override the parameters by using the interface provided by the parameter system.

If the driver publishes the instParamModify method and the middleware changes a parameter value, the driver is notified of the change.

The middleware can retrieve the parameter value at any time by using the name of the parameter.

The device driver can, at any time, retrieve the parameter value by specifying either the name or an index of the parameter table.
13.2.2 Using the Parameter System

This section discusses the parameter system data structures and the interfaces that are available for use by the BSP, device drivers, and middleware.

Data Structures

The following data structures are used in the parameter system:

**VXB INST PARAM VALUE**

VXB INST PARAM VALUE can be of type void *, UINT32, UINT64, char *, or FUNCPTR. This parameter is used for storing the parameter value of the specified type. Usage is as follows:

```c
typedef union vxbInstParamValue VXB_INST_PARAM_VALUE;
union vxbInstParamValue
{
    void * pValue;
    UINT32 int32Val;
    UINT64 int64Val;
    char * stringVal;
    FUNCPTR funcVal;
};
```

**VXB PARAMETERS**

VXB PARAMETERS contains the details of a given parameter including the name of the parameter, type of the parameter, and the associated value. A table of such data structures is used by the device driver to provide the parameter default values. Usage is as follows:

```c
typedef struct vxbParams VXB_PARAMETERS;
struct vxbParams
{
    char * paramName;
    UINT32 paramType;
    union vxbInstParamValue value;
};
```

**VXB INST PARAM OVERRIDE**

VXB INST PARAM OVERRIDE is the data structure used by a BSP to override a parameter value.

Such an entry should be available for every parameter the BSP wishes to override. The BSP should specify the instance name, the unit number of the instance, the parameter name, the parameter type, and the value that needs to be updated.
Usage of this data structure is as follows:

typedef struct vxbInstParamOverride VXB_INST_PARAM_OVERRIDE;
struct vxbInstParamOverride
{
    char * instName;
    int unit;
    char * paramName;
    UINT32 paramType;
    union vxbInstParamValue value;
};

13.2.3 Parameter System Interfaces

This section discusses the interfaces provided by the VxBus parameter system.

vxbInstParamSet( )

The vxbInstParamSet( ) routine is used by the middleware to update a parameter of the instance. Usage is as follows:

STATUS vxbInstParamSet
{
    VXB_DEVICE_ID pInst,
    char * paramName,
    UINT32 paramType,
    VXB_INST_PARAM_VALUE * value
};

vxbInstParamByNameGet( )

The vxbInstParamByNameGet( ) routine is used by the middleware and the device driver to get the parameter value using the name of the parameter as input. Usage is as follows:

STATUS vxbInstParamByNameGet
{
    VXB_DEVICE_ID pInst,
    char * paramName,
    UINT32 paramType,
    union vxbInstParamValue * pValue
};

vxbInstParamByIndexGet( )

The vxbInstParamByIndexGet( ) routine is used by the device driver to get the parameter value using the index into the parameter table as input.
13.3 Timer Abstraction Routines

VxBus provides a set of routines that can be used by device drivers or middleware to introduce a busy wait to a system. The busy wait can have millisecond or microsecond granularity.

The following routines are provided for this purpose:

vxbDelay( )

The vxbDelay() routine introduces a busy wait of one millisecond. Usage is as follows:

```c
void vxbDelay (void);
```

vxbMsDelay( )

The vxbMsDelay() routine introduces a busy wait of delayTime milliseconds. Usage is as follows:

```c
void vxbMsDelay
(int delayTime);
```

vxbUsDelay( )

The vxbUsDelay() routine introduces a busy wait of delayTime microseconds. Usage is as follows:

```c
void vxbUsDelay
(int delayTime);
```
14.1 Introduction

The hEnd model divides the END driver into two levels: an upper level that interfaces with a protocol layer, such as IP, and a lower level that handles all hardware-specific accesses. The levels are described as follows:

- system level (SL)
- device-specific level (DL)

The SL is provided by Wind River. The DL is created by the driver developer. This separation establishes a relationship between the SL and the DL where the SL interfaces with the layers above the driver and manages the device abstraction.
while the DL controls a specific device. Because of this, the SL is not concerned with the particulars of a given device's operation.

The SL and the DL communicate with each other through an API specified for their interface, and the DL is managed by the SL through this interface. The DL makes all device-specific accesses to the device's hardware or descriptors. In this way, the DL abstracts a device's hardware, hiding the particulars of its operation.

The hEnd driver model:

- Simplifies the responsibilities of the driver developer by requiring a smaller amount of code to be written for each new hEnd driver.

- Provides operational consistency by reusing the SL; this guarantees that all hEnd drivers use the same proven design for all but the DL code.

- Provides operational consistency by using the reusable hEnd; this guarantees that all hEnd drivers use the same proven design.

- Facilitates upgrade and fix deployment. The DL code is separated from the SL code into separate files. The two are only combined at compile time. Therefore, fixes and upgrades to the SL code can be deployed across all hEnd drivers by recombining the DL files for each driver with the new fixed or upgraded SL file and recompiling.

- Reduces the learning curve for engineers new to hEnd drivers because understanding of the SL operation provides a basis for developing multiple DLs.

- Allows all of the files included in the devHEnd.c driver file are co-compiled with the SL. Token merging converts the SL tokens to match the DL specification. The whole driver fits together as if it were written in a single file. Logical modularity vanishes into a single object and there is no data hiding between the DL and the SL. Any construct or local defined by one layer is visible to the other.

- Adds additional functionality by defining additional routines or including additional standardized modules supporting token merging to existing DL files.
14.2 Architecture Overview

The hEnd driver as a whole consists of three logical components: command and control, a receiver, and a transmitter. The command and control is that part of the driver that interacts with the system for configuration, initialization, filter and register setting and alteration, and interrupts. The receiver is that part of the driver that provides a conduit for packets to flow from the external connection into the driver and up to the protocol above the driver. The transmitter is that part of the driver that provides a conduit for packets to flow from the protocol above the driver out through the external connection.

All three components are separated into SL and DL layers. The hEnd model is based on separating the hardware-specific aspects and the reusable general purpose aspects that most END drivers share in common (for more information on END drivers, see 4. END Ethernet Drivers). The point of separation between each component’s SL and the DL is the point where specific knowledge of the device register set or descriptor layout is required. Above the separation point, general-purpose code manages an abstraction of the device. Below the separation point, the code operates according to the specification for a particular hardware device.

The hEnd driver model distinguishes between performance-critical sections of the driver and non-performance-critical areas. The performance critical sections are those sections where execution times affect the efficiency of packet reception or transmission.

In performance critical sections of the driver, the SL must call the DL to access descriptors or registers. Standard function calls across the boundary are a performance detriment. To eliminate performance degradation across the boundary in critical sections of code, hEnd specifies that the DL functions called by the SL in performance-critical sections must be declared inline. This subset of inline DL functions is specified in the DL API. The compiler allows only small functions to be inlined. To ensure inlining of DL functions, these functions must be as small as possible. In addition, compiler flags can be used to adjust the allowed amount of code.

Underlying the hEnd driver is the VxBus framework (see 10. VxBus). VxBus provides methods to abstract the CPU architecture, bus type, and relative location of devices for accessing hardware and DMA memory. It allows for device drivers to use a standardized API to access hardware and obtain DMA-safe memory.
The major functional components of the hEnd driver model are shown in Figure 14-1.

In Figure 14-1, note that the API/control arrows below the SL receiver and the SL transmitter meet point-to-point in the DL receiver and DL transmitter. This is also true between the bus subsystem and the DL receiver and transmitter. This signifies that the function calls these arrows represent may be inline. The command and control arrows do not meet point-to-point, signifying that the command and control function calls are not inline.
14.2.1 Command and Control

The SL command and control module manages the overall driver. This includes creating the driver END_OBJ structure, managing the driver’s configuration and initialization, initializing the driver MIB interface, and managing the driver ISRs. In addition to these responsibilities, command and control must also provide the driver ioctl() functionality. Command and control starts and stops the driver software as necessary and manages all driver load and unload operations.

DL command and control manages an abstraction of the device register set by controlling the device under the management of the SL command and control. The DL command and control sets the device hardware to a quiescent state and configures the device in the prescribed modes. In addition, DL command and control is responsible for starting and stopping the device hardware and for providing service routines for each device interrupts.

14.2.2 Receiver

The SL receiver manages the DL receiver by parsing the configuration parameters passed to it from SL command and control. The SL receiver also manages the configuration and initialization of the DL receiver and the receive algorithm. The SL receiver adjusts MIB objects and counters in pushed mode by interfacing with the MIB. It also receives packets from the DL and passes them to the MUX.

The DL receiver abstracts and manages the devices receive mechanism by controlling the device receiver under the management of the SL receiver. It configures and initializes the device’s receiver DMA engine and provides abstracted, inline methods for perusing and manipulating the device’s receiver DMA so that the SL receiver can manage the device’s receive DMA ring.

14.2.3 Transmitter

The SL transmitter manages the DL transmitter by configuring and initializing the device’s receiver DMA engine and by parsing the configuration parameters passed to it from SL command and control. The SL transmitter manages the configuration and initialization of the DL transmitter as well as the transmit algorithm. It adjusts MIB objects and counters in pushed mode by interfacing with the MIB. It also receives packets from the MUX and passes them to the DL transmitter.

The DL transmitter abstracts the device transmit mechanism by controlling the device transmitter under the management of the SL transmitter. It configures and
initializes the device’s transmit DMA engine. The DL transmitter provides abstracted, inline methods for perusing and manipulating the device’s transmit DMA queue so that the SL transmitter can manage the queue.

### 14.2.4 VxBus Hardware Access Abstraction

The VxBus framework abstracts hardware access by:

- Providing guaranteed-correct methods for accessing registers and memory relative to the device.
- Providing the option for optimized inline access methods.

For more information, see 11.4 Device Access in the Register Device Model, p.301.

### 14.3 Driver Files

The hEnd driver is composed of several files that accommodate the two-layered design (SL and DL). These files include:

- `installDir/vxworks-6.x/target/src/hwif/hEnd/dev.c`
  Device-level (DL) source file; contains the functions required by the SL.

- `installDir/vxworks-6.x/target/src/hwif/hEnd/devHEnd.c`
  Driver configlette. Currently, this file only includes the DL source file and is referenced in the `CONFIGLETTES` attribute of the driver component.

- `installDir/vxworks-6.x/target/src/hwif/hEnd/hEnd.c`
  System-level (SL) source file; this file should only be used for reference, and not modified.

- `installDir/vxworks-6.x/target/src/hwif/hEnd/hEndVxBus.c`
  Contains the structures and functions needed to interface the hEnd driver with the VxBus framework.

- `installDir/vxworks-6.x/target/src/hwif/h/hEnd/devHEnd.h`
  Device-level header file; contains the `DRV_CTRL` structure, macros required for the SL, and device-specific macros and structures.
14 Hierarchical END Drivers (hEND)

14.4 Driver Configuration

installDir/vxworks-6.x/target/src/hwif/h/hEnd/hEnd.h

System-level (SL) header file. This file should only be used for reference, and not modified.

installDir/vxworks-6.x/target/config/comps/vxWorks/40devHEnd.cdf

Contains the driver component definition.

14.3.1 File Integration

The files listed previously, with the exception of the CDF file, are all compiled into a single translation unit. The generic files hEnd.c, hEndVxBus.c, and hEnd.h contain variable, routine, and macro names that are altered during preprocessing using a standard C technique called token merging. The variable, routine, and macro names in the generic files initially contain placeholder strings that are replaced at compile time. The DL header file must define unique replacement strings in order to ensure that there are no symbol conflicts when multiple hEnd drivers are built into the same VxWorks image.

14.4 Driver Configuration

hEnd drivers, unlike END drivers, are visible components in projects. Each driver includes a CDF file (for information on CDF files, see the VxWorks Programmer’s Guide: Kernel). This means that they can be configured into VxWorks using Workbench or the vxprj command line facility. An hEnd component supports the ability to configure several instances of the same driver in the same VxWorks image.

Configuration parameters in an hEnd driver are of two general types:

- Required Parameters—Externally provided parameters required by the hEnd driver from an external source such as by a BSP or CDF.
- Optional Parameters—Parameters established by default by the driver but capable of being overridden from external sources.
14.4.1 Configuration with CDF

Configuration parameters can be added to the hEnd driver CDF file just like other VxWorks components. However, these parameters will affect all instances of the hEnd driver. The tables in hwconf.c allow for more flexibility.

14.4.2 Configuration in hwconf.c

Each VxBus-compliant BSP includes a file called hwconf.c. This file contains structures that describe the general bus layout of the system and provide for passing required and optional parameters. Required parameters are passed to the drivers through hcfResource arrays. The hwconf.c file must have an hcfResource array for each configured instance of an hEnd driver that resides on the processor local bus (PLB). Optional parameters can be passed through the sysInstParamTable, also found in the hwconf.c file.

For more information, see 11.3.2 PLB Registration, p.300.

14.4.3 Default Configuration

The driver DL must provide a pointer to a table containing the driver default values for each parameter of interest. This is supplied to VxBus during driver registration. Each parameter consists of name, type, and default value. In the case of the hEnd driver, these parameters must include at minimum:

- An initialized HEND_RX_QUEUE_PARAM structure for each configured receive queue. (Defined in installDir/vxworks-6.x/target/src/hwif/h/hEnd/hEnd.h)
- An initialized HEND_TX_QUEUE_PARAM structure for each configured transmit queue. (Defined in installDir/vxworks-6.x/target/src/hwif/h/hEnd/hEnd.h)
- Any other parameters relevant to the driver DL, which may be overridden by the BSP or an application.

At any time after the instance is created, middleware is able to specify parameters for the instance by calling the vxbInstParamSet() routine. If middleware tries to set a parameter before the instance is created, the operation fails. If middleware tries to set a parameter after the instance is initialized, the operation succeeds, but the driver ignores the set in most cases.
The BSP can provide instance override values for any parameter for any instance using the VXB_INST_PARAM_OVERRIDE sysInstParamTable[] defined in the BSP hwconf.c file. This table can be loaded with pointers to device- and instance-specific parameters for VxBus to use during device instantiation.

14.4.4 Configuration from an Application

In some cases you may want a particular instance of an hEnd driver to tailor its configuration for an application. For example, you may want to have a task level routine run in a context other than the default tNetTask context. This only applies to default parameters, externally provided parameters can only be set in the hwconf.c file or in a CDF.

Interrupt Configuration

A network interface device can generate an interrupt for each of the interrupt events it supports. Network devices universally support at least three interrupt events.

- receive
- packet transmit complete
- error

The number of external interrupt lines that devices can connect to varies from one device to another. Some devices have a single interrupt line and generate interrupts for all events through the same interrupt line, other devices have a discrete interrupt line for each interrupt event. A device driver must provide a separate ISR for each external interrupt line a device is connected to. Each ISR is dedicated to servicing interrupts for its particular interrupt line. An ISR handles all events that generate interrupts on its particular interrupt line. In VxWorks, ISRs schedule task level routines to service conditions that cause interrupt events. The hEnd model uses job queues supported in jobQueueLib to provide flexibility in scheduling task-level handlers in contexts other than tNetTask. This allows the task level service of events to be prioritized according to system configuration considerations.
14.4.5 Configuring a Task Level Routine to Run in a Non-default Context

To accomplish the example in 14.4.4 Configuration from an Application, p.331 (that is, an application configuring an hEnd driver so a task level routine runs in a non-default context), you must have a default parameter that can be overridden which controls what context a task level routine runs in. This discussion assumes that you want to configure an instance’s receive handler to run in an application specified context. The mechanism—which allows applications to create a context for a driver’s receive handler to run in—is the job queue. The job queue used by a specific instance’s receive handler is specified by the jobQueId field in the HEND_RX_QUEUE_PARAM structure. The values in HEND_RX_QUEUE_PARAM are initially set by the hEnd driver when it sets all of the default values for configuring the receive handler. Applications can pass an alternative HEND_RX_QUEUE_PARAM into an hEnd driver using the vxbInstParamSet() routine with the jobQueId field initialized. To obtain a JOB_QUEUE_ID, the application calls jobQueueCreate() which returns a JOB_QUEUE_ID. Note that jobQueueCreate() does not spawn a task to process queue jobs. This can be done in the application by spawning a task to execute jobQueueProcess() on the returned job queue ID. (For more information, see installDir/vxworks-6.x/target/src/wrn/coreip/common/daemon/jobQueueLib.c)

14.5 Loading an hEnd Driver

Each hEnd driver is required to register a muxDevConnect method with VxBus. The driver’s muxDevConnect method is called by VxBus during the driver initialization. The driver’s muxDevConnect method calls muxDevLoad() with the driver’s unit number and the driver’s devEndLoad() routine.

The MUX issues devEndLoad() to load the hEnd driver which maintains backward compatibility with the legacy model. In the legacy model, END drivers obtain their parameters through devEndLoad() using an END_LOAD_STRING. The devEndLoad() routine furnishes the MUX with the driver’s name. In the VxBus model, hEnd drivers no longer use the END_LOAD_STRING to receive their parameters. However, the devEndLoad() routine still furnishes the MUX with the driver’s name. Therefore, devEndLoad() is required to maintain backward compatibility with the MUX.
14.6 Unloading an hEnd Driver

The unload routine in an hEnd driver can only be called using `muxUnload()`. Before the `muxUnload()` routine calls the driver unload routine, it must unbind the device driver from any protocols to which it is bound. The driver unload routine must then complete the unload by:

- Disabling the device. That is, by disabling interrupts and unbinding the device from the MUX.
- Freeing its associated memory. This is accomplished by:
  - Flushing job queues.
  - Relinquishing semaphores, descriptors, and tuples.
  - Returning memory to the driver pool.
  - Freeing the driver control structure.

14.7 Developing a New Driver

In order to support a new device in the hEnd model, a set of DL files must be created. At a minimum, this set must include a DL header (`devHEnd.h`) and a DL source file (`devHEnd.c`). These two files must contain all the routines and macros required by the SL, and should contain all device-specific information like register offsets and any special code required for operation.

Even though DL files are written for specific devices, they should be written in a CPU architecture agnostic format using coding techniques that work regardless of the target CPU architecture. With the advent of VxBus, register access is made simple from the driver standpoint, as the VxBus facility abstracts the series of bus translations that may exist between the CPU and the device.

VxBus provides hardware access methods and guarantees that all accesses through the provided methods are correct relative to the location of the device in the physical system. VxBus optionally provides a mechanism for the BSP to provide the hardware access methods in optimized form. To take advantage of the BSP provided optimized access macros, the hEnd driver needs to be compiled in the BSP. This allows the driver and the BSP to be compiled in the same translation unit,
which in turn allows the optimized macros to be inlined in the driver. This
eliminates function calls at performance critical junctures.

14.7.1 The Development Process

This section provides a suggested process for writing a new DL using the template
driver files, which can be found on the online support Web site. These files are
referred to in this guide as tmplt.c and tmpltHEnd.h. The process outlined below is
only a guide, it is not a comprehensive procedure.

Step 1: Research the Hardware and Driver Model

Gather the device hardware manual, and if available, older END or other drivers
for reference. Read this chapter to understand the hEnd model.

Step 2: Write the CDF File

The easiest way to include your driver in a VxWorks image is using the project
facility. Use the template CDF file, 40tmpltHEnd.cdf, or the CDF file for an existing
hEnd driver as a guide.

Step 3: Write the DL Header File

Start with tmpltHEnd.h. In this file, translate the register specifications in the
hardware manual into macros representing register offsets, bitfields, and values.
Be sure to replace all instances of tmplt with your device name prefix.

NOTE: In the template, the device name is tmplt. For variable and function
symbols, the naming convention will be tmplt + Object + Action, as in
tmpltIntEnable(). Similarly, the macro naming conventions includes the
device-specific prefix TMPLT. Choose an appropriate short (between 2 and 7
characters) prefix for your symbol and macro names.

Step 4: Write the DL source file

The DL file contains the main includes, such as network headers, and also includes
other hEnd files.

First, replace the example device name, tmplt, and TMPLT in the template file with
the name of your device.

Next, fill in the required routines. The bodies of these routines should consist
mostly of register accesses, hints about what to put into these routines are included
in the template. It is not expected that the entire driver be written in one pass. Wind River recommends the following process:

1. Fill out the `tmpltDLInit()` routine. This routine initializes all device-specific fields in the `TMPLT_DRV_CTRL` structure, and puts the device into a quiescent state. Because many of these actions may be necessary to restart a device once it is stopped, it may be desirable to collect them in a new routine, or set of routines, that can be called in `tmpltDLInit()` and elsewhere.
   - A dump of the device registers can be used to determine if this routine is correctly initializing the device.
   - In the process, correctly functioning register access methods must be written. The macros included in the template should work with little or no modification.

2. Fill out the descriptor initialization routines, `tmpltRxDescInit()` and `tmpltTxDescInit()`. In both cases, ready the specified descriptor for operation. This may involve clearing or setting a specific status in a status field. Some devices may require setting a wrap bit in the last descriptor. `tmpltRxDescInit()` is slightly more complicated; its last parameter is a pointer to a buffer that the SL allocates for this descriptor. Perform the appropriate address translation and program the address field in the descriptor.
   - At this point, it may be helpful to write a simple routine to dump the fields of all the descriptors. This ensures that they are set up correctly.

3. Write a simple ISR and fill out the `tmpltIntConfig()` routine so that it references this ISR. This allows the ISR to be connected by the SL. Have the ISR check for a receive interrupt. Ensure the ISR is called by using the `logMsg()` routine.

4. Extend your ISR so that it posts a job to the receive job queue, when a packet is received. Check that `__RxHandle()` is called. Remember to set the busy flag in your ISR after the job is posted, and to disable the receive interrupt to prevent queuing multiple receive jobs. See the template DL file for an example. The busy flag is cleared in the receive handler.

5. Finish the receiver by placing receive descriptor management code in the routines `tmpltRxDescStatusGet()` and `tmpltRxDescStatusClear()`.

6. Fill out the DL transmit routine `tmpltTxPktSend()`. This routine should enqueue the passed-in `M_BLK` chain into the device’s descriptors, setting descriptor fields and flags as appropriate. Keep track of the number of descriptors used—it may differ from the number of fragments—and return this number to the caller. This is essential to maintain consistency between the
device and driver transmit rings. Some devices may require setting a particular register to commence transmission.

7. Extend your ISR or fill in the tmpl/TxInt() ISR to handle transmit complete and/or other transmit interrupts. In the hEnd model, transmit interrupts are handled at the task-level in tNetTask, therefore be sure to include a call to netJobAdd(), passing it the function pointer __(_(TxHandler)). The transmit interlock flag is located in pDrvCtrl->hEnd.txHandlerQued. Be sure to set this to TRUE. It is reverted to FALSE in the transmit handler.

8. Complete the transmitter by filling in the tmpl/TxDescStatusGet() routine. This routine should check the appropriate transmit descriptor fields and determine if the device still owns the descriptor.

At this point, the driver should exhibit basic functionality.

14.7.2 Additional Tasks

Once the driver is functioning at a basic level, besides extending its capabilities there are a few additional development tasks that should be considered.

Optimized Access Macros

The access functions provided by VxBus are guaranteed to work correctly, but the performance may not be optimum. The recommended action is to write optimized access macros in the BSP. The template driver is already set up to accept two BSP-defined access macros that override the default VxBus access functions: TMPLT_REGISTER_READ32 and TMPLT_REGISTER_WRITE32.

Command-Line Build Capability

hEnd drivers do not compile correctly in the BSP unless two requirements are met:

- First, a devHEnd.o must be compiled. Edit installDir/vxworks-6.x/target/src/hwif/hEnd/Makefile so that your driver is compiled.
- Second, the file installDir/vxworks-6.x/target/src/hwif/util/cmdLineBuild.c must reference your driver.

Note that in the BSP command-line build, the optimized access macros mentioned above are not available. These are only available when the driver is compiled with the project. There may be some cases, however, when a .o is preferable.
Parameter System Integration

By adding parameters to the default parameter table in the DL file, it becomes possible for a BSP or other software entity to customize the driver to suit various scenarios. The DL template contains a stub table that can be filled out. Additionally, the DL routine `tmpltDLParamParse()` must be extended to query VxBus for the value of each of the driver’s parameters.
Symbols

__(RxHandle)( ) 335
__(TxHandler)( ) 336
_pLinkPoolFuncTbl 66, 67

Numerics

00tffs.cdf 156
10vxBus.cdf 313
4.3 BSD Ethernet drivers
  porting to the END driver model 139
  upgrading to a 4.4 BSD Ethernet driver 135
4.4 BSD Ethernet drivers
  porting to the END driver model 139
40devHEnd.cdf 329
40tmpltHEnd.cdf 334

A

access macros 16
access routine 278
accessing registers through VxBus 22
address resolution, arpreolve( ) 53

AMD/Fujitsu flash support
  CFI 146
  non-CFI 147
  amdmtd.c 147
analyzing your driver for cache problems 28
APIs, protocol to MUX 47
application
  hEnd configuration from 331
arpcom 133
arpreolve( ) 53
association list 64
ataDrv.c 264
ATAPI drivers 263

B

BAM 169
block allocation map 143, 169
see also BAM
boot block 146
BSD 4.3 driver model 39, 131
BSD 4.4 driver model 131
BSD Ethernet drivers 131
BSPs
adding drivers 19
required BSP support 20
config.h 61, 157
routines
sysHwInit2() 12
sysScsiInit() 224
run-time initialization of cacheLib 31
BUFFER_WRITE_BROKEN 145
building multiple hEnd drivers into a VxWorks image 329
bus 278
discovery 278
match 278
snooping 31, 33
support 10
  legacy (non-VxBus) drivers 11
  VxBus bus controller device drivers 12
  VxBus-compliant drivers 11
type 278
bus controller 278
device drivers 315
devices 280
busID 298, 299
busVer 299

C

cache
coherence 26
considerations 26
driver attributes 28, 30
flush and invalidate macros 32
library 27, 31
see also cacheLib
CACHE_DMA_FLUSH 33
CACHE_DMA_INVALIDATE 33
CACHE_DMA_IS_WRITE_COHERENT 34
CACHE_DMA_PHYS_TO_VIRT 34
CACHE_DMA_VIRT_TO_PHYS 34
CACHE_DMA_xxxx 28, 29
CACHE_DRV_FLUSH 34
CACHE_DRV_INVALIDATE 34
CACHE_DRV_xxxx 28, 29, 33
CACHE_PIPE_FLUSH 32
CACHE_USER_FLUSH 33
CACHE_USER_INVALIDATE 33
CACHE_USER_xxxx 28, 29, 33
cacheDmaMalloc() 27, 28, 29, 33, 34
cacheLib 26, 27, 31
cacheLib.h 27
CDF 19, 21
  hEnd driver configuration 330
CDL 21
CFI/SCS flash support 145
CFI_DEBUG 145
cfamd.c 144
cfiscs.c 144, 145
checksum offloading 88
child 278
cmdLineBuild.c 313, 336
commonCmdsTest() 230
component description file
see CDF
component description language
see CDL
config.h 61
configNet.h 59
CSUM_DATA_VALID 91
CSUM_DELAY_DATA 91
CSUM_DELAY_DATA6 91
CSUM_DELAY_IP 91
CSUM_FRAGMENT 91
CSUM_IP 91
CSUM_IP_CHECKED 91
CSUM_IP_FRAGS 91
CSUM_IP_HDRLEN 91
CSUM_IP_VALID 91
CSUM_PSEUDO_HDR 91
CSUM_RESULTS 91
CSUM_TCP 91
CSUM_TCP_SEG 91
CSUM_TCPv6 91
CSUM_TCPv6_SEG 91
CSUM_UDP 91
CSUM_UDPv6 91
CSUM_XPORT_CSUM_OFF 91
CSUM_XPORT_HDRLEN 91
D

data structures
  in the VxBus parameter system  319
DEBUG_PRINT  145
debugging
  VxBus debug messages  313
  VxBus device drivers  312
design guidelines  12
dev.c  328, 333
DEV_OBJ  99
developing
  hEnd drivers  334
devEndLoad()  332
devHEnd.c  328
devHEnd.h  328, 333
devHEnd.o  336
device  278
  discovery in VxBus  280
  enumeration  316
  insertion and removal with VxBus  281, 282
device drivers
  multi-function devices  310
  VxBus-compliant  297
DEVICE_WRITES_ASYNCRONOUSLY  30, 33
devID  299
DEVMETHOD  288
devProbe  299
direct memory access  28
  see also DMA
directCmdsTest()  230
directRwTest()  230
disk-on-chip support  148
DL  323
DMA  28
do_protocol()  137
do_protocol_with_type()  46, 133, 137
document terminology  3
documentation, driver standards  13
downloadable driver support  35
downstream  278

driver  279
  cache attributes  28
  configuration
    hEnd  329
    hEnd default  330
documentation standards  13
  hEnd model  323
  naming conventions  13
driver cache attributes  30
driver methods  279, 281, 282
  instParamModify  318
  muxDevConnect  332
  VxBus  288
  DRV_CTRL  137, 315, 328
drvName  299, 300

E

EAGAIN  114
EIOCGADDR  109
EIOCGBUF  110
EIOCGFLAGS  109
EIOCGFCAP  92
EIOCGMIB2  110
EIOCGPOLLCONF  110
EIOCGPOLLSTATS  110
EIOCMULTIADD  109
EIOCMULTIDEL  109
EIOCMULTIGET  109
EIOCPOOLLSTART  109
EIOCPOOLLSTOP  109
EIOCSADDR  109
EIOCSFLGS  109
EIOCSFCAP  92
END driver  37

  adding a multicast address  114
  adding drivers to VxWorks  59
  association list  64
  backwards compatibility  121
  control structure  61
  deleting a multicast address  115
  driver responsibilities  44
  entry points  40
  error conditions  122
fair access bounding 70, 73
forming an address for packet transmission 116
getting a data-only mBlk 117
getting the multicast address table 115
handling a polled receive 113
handling a polled send 112
implementing the generic MIB interface 123
interface to VxWorks 54
interrupt handlers 56
interrupt masking 58
interrupt re-enabling 70, 77
launching your driver 54
loading a device 104
mBlk structure 101
memory resources 61
MUX responsibilities 44
network layer to data link layer address resolution 53
performance 121
protocol responsibilities 44
providing a control interface 108
receive and transmit descriptors 61
receive handler interlocking flag 70, 74
receive loop 70, 71
receiver stall handling 70, 76
required entry points 102
required structures 96
returning addressing information 117
sending data out on the device 110
setting up a memory pool 66
starting a loaded driver 111
status dump routines 129
stopping a loaded driver 111
support for scatter-gather 82
transmit descriptor clean routine 84
transmit-packet-complete handler interlocking flag 82
two-tiered polling 70, 77
unloading a device 106
END driver components 42
END_CAPABILITIES 89, 92
END_ERR_BLOCK 123
END_ERR_DOWN 51, 123
END_ERR_FLAGS 123
END_ERR_INFO 51, 123
END_ERR_NO_BUF 123
END_ERR_RESET 51, 123
END_ERR_UP 51, 123
END_ERR_WARN 51, 123
END_IFCOUNTERS 128
END_IFDRVCONF 128
END_LOAD_FUNC 60
END_LOAD_STRING 60, 332
END_OBJ 96
END_OBJ_INIT 99
END_RCV_RTN_CALL 73
endAddressForm() 103, 116
dendDevTbl[] 59
dendEtherPacketAddrGet() 117
dendIoctl() 103, 108, 139, 140
dendLoad() 54, 60, 80, 96, 100, 103, 104, 139
dendM2Free() 125
dendM2Init() 124
dendM2Ioctl() 125
dendM2Packet() 125
dendMCastAddrAdd() 98, 103, 114, 139
dendMCastAddrDel() 98, 103, 115, 139
dendMCastAddrGet() 98, 103, 115, 139
dendPacketAddrGet() 104
dendPacketDataGet() 104, 117
dendPollReceive() 103, 113, 139
dendPollSend() 103, 112, 139
dendReceive() 69
dendSend() 103, 110, 139, 140
dendStart() 103, 111
dendStop() 103, 111
dendTbl 60
dendUnload() 103, 106, 139
enhanced network driver see END driver enumeration device 316
erase units 178
ether_attach() 135
ether_output() 133
etherhooks 134
etherInputHook() 46, 134
etherMultiLib 114
Index

Ethernet driver  37
  see also END driver
etherOutputHook()  46, 134

F
files
  10vxBus.cdf  313
  40devHEnd.cdf  329
  40tmpltHEnd.cdf  334
  cmdLineBuild.c  313, 336
  dev.c  328, 333
  devHEnd.c  328
  devHEnd.h  328, 333
  devHEnd.o  336
  hEnd driver  328
  hEnd.c  328
  hEnd.h  329, 330
  hEndVxBus.c  328
  hwconf.c  300, 330
  jobQueueLib.c  332
  tmplt.c  334
  tmpltHEnd.h  334
flash
  device layout  178
  erase units  178
flash translation layer  143, 166
  see also FTL
  FLASH_BASE_ADRS  161
  FLASH_SIZE  161
  flbase.h  151
  flDelayMsec()  160
  flDontNeedVpp()  153
  FLFlash  149
  flflash.h  149
  flNeedVpp()  153
  flSetWindowSize()  161
  FLSocket  159
  flSocketOf()  163
  fsysystem.h  155
  flWriteProtected()  153

FTL
  overview  171
  structures  171
  terminology  166

H
hardware timers, characteristics  236
hcfResource  330
hEnd  323
  architecture overview  325
  configuration in hwconf.c  330
  configuration in the application  331
  configuration with CDF  330
  default configuration  330
  developing a new driver  333
  development process  334
  device-specific level  323
  driver configuration  329
  driver files  328
  file integration  329
  interrupt configuration  331
  job queues  332
  loading a driver  332
  naming convention for
    routines and macros  334
    optimized access macros  336
    system level  323
    transmit interrupts  336
    unloading a driver  333
    VxBus framework  325
hEnd.c  328
hEnd.h  329, 330
  HEND_RX_QUEUE_PARAM  330, 332
  HEND_TX_QUEUE_PARAM  330
  hEndVxBus.c  328
  hotswap  281, 282
    support  35
  hwconf.c  300
    hEnd driver configuration  330
  hwMemAlloc()  315
I

I/O mapped systems 9
I28F008.c 147
I28F016.c 146
IF_ENQUEUE 134
IFCAP_CAP0 90
IFCAP_CAP1 90
IFCAP_CAP2 90
IFCAP_IPCOMP 90
IFCAP_IPSEC 90
IFCAP_JUMBO_MTU 90
IFCAP_NETCONS 90
IFCAP_RXCSUM 90
IFCAP_RXCSUMv6 90
IFCAP_TCPSEG 90
IFCAP_TCPSEGv6 90
IFCAP_TXCSUM 90
IFCAP_TXCSUMv6 90
IFCAP_VLAN_HWTAGGING 90
IFCAP_VLAN_MTU 90
IFF_ALLMULTI 97
IFF_BROADCAST 97
IFF_DEBUG 97
IFF_LINK0 98
IFF_LINK1 98
IFF_LINK2 98
IFF_LOOP 98
IFF_LOOPBACK 97
IFF_MULTICAST 98
IFF_NOARP 97
IFF_NOTRAILERS 97
IFF_OACTIVE 97
IFF_POINTOPOINT 97
IFF_PROMISC 97
IFF_RUNNING 97
IFF_SCAT 98
IFF_SIMPLEX 97
IFF_UP 97
INCLUDE_END 61
INCLUDE_MTD_AMD 148
INCLUDE_MTD_CFISCS 145
INCLUDE_MTD_I28F008 147
INCLUDE_MTD_I28F016 147
INCLUDE_NET_INIT 59

INCLUDE_NETWORK 59
INCLUDE_TIMESTAMP 239
INCLUDE_USER_TIMESTAMP 239
initialization sequence
VxBus 282, 283, 298
instance 279
instParamModify 318
intConnect( ) 15, 26, 292
integrating
driver code with VxBus 313
drivers with the project facility 20
Intel 28F008 flash support 147
Intel 28F016 flash support 146
interrupt controller drivers 264
interrupt controllers 12
interrupt handlers 56
interrupt masking 58
interrupt service routines
see ISRs
interrupts
difference between VxBus
and kernel interrupts 292
VxBus 292
intLib 292
intLock( ) 8
ISRs 15

J

job queues
hEnd 332
JOB_QUEUE_ID 332
jobQueueCreate( ) 332
jobQueueLib 331
jobQueueLib.c 332
jobQueueProcess( ) 332

L

libraries
etherMultiLib 114
intLib 292
jobQueueLib 331
netBufLib 55, 65
SCSI 190
scsi2Lib 189
scsiCommonLib 189, 193
scsiCtrlLib 192
scsiDirectLib 189, 192
scsiMgrLib 190
scsiSeqLib 190, 193
LL_HDR_INFO 100
loading an hEnd driver 332
logMsg( ) 25, 26, 335

**M**

macros

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>16</td>
</tr>
<tr>
<td>BUFFER_WRITE_BROKEN</td>
<td>145</td>
</tr>
<tr>
<td>CACHE_DMA_FLUSH</td>
<td>33</td>
</tr>
<tr>
<td>CACHE_DMA_INVALIDATE</td>
<td>33</td>
</tr>
<tr>
<td>CACHE_DMA_IS_WRITE_COHERENT</td>
<td>34</td>
</tr>
<tr>
<td>CACHE_DMA_PHYS_TO_VIRT</td>
<td>34</td>
</tr>
<tr>
<td>CACHE_DMA_VIRT_TO_PHYS</td>
<td>34</td>
</tr>
<tr>
<td>CACHE_DMA_xxxx</td>
<td>28, 29</td>
</tr>
<tr>
<td>CACHE_DRV_FLUSH</td>
<td>34</td>
</tr>
<tr>
<td>CACHE_DRV_INVALIDATE</td>
<td>34</td>
</tr>
<tr>
<td>CACHE_DRV_xxxx</td>
<td>28, 29, 33</td>
</tr>
<tr>
<td>CACHE_PIPE_FLUSH</td>
<td>32</td>
</tr>
<tr>
<td>CACHE_USER_FLUSH</td>
<td>33</td>
</tr>
<tr>
<td>CACHE_USER_INVALIDATE</td>
<td>33</td>
</tr>
<tr>
<td>CACHE_USER_xxxx</td>
<td>28, 29, 33</td>
</tr>
<tr>
<td>CFI_DEBUG</td>
<td>145</td>
</tr>
<tr>
<td>DEBUG_PRINT</td>
<td>145</td>
</tr>
<tr>
<td>DEVMETHOD</td>
<td>288</td>
</tr>
<tr>
<td>END_OBJ_INIT</td>
<td>99</td>
</tr>
<tr>
<td>END_RCV_RTN_CALL</td>
<td>73</td>
</tr>
<tr>
<td>flush and invalidate cache</td>
<td>32</td>
</tr>
<tr>
<td>IF_ENQUEUE</td>
<td>134</td>
</tr>
<tr>
<td>INCLUDE_MTD_AMD</td>
<td>148</td>
</tr>
<tr>
<td>INCLUDE_MTD_CFI_3SCS</td>
<td>145</td>
</tr>
<tr>
<td>INCLUDE_MTD_I28F008</td>
<td>147</td>
</tr>
<tr>
<td>INCLUDE_MTD_I28F016</td>
<td>147</td>
</tr>
<tr>
<td>INCLUDE_NET_INIT</td>
<td>59</td>
</tr>
<tr>
<td>INCLUDE_NETWORK</td>
<td>59</td>
</tr>
<tr>
<td>naming</td>
<td>17</td>
</tr>
<tr>
<td>optimized register access</td>
<td>304</td>
</tr>
<tr>
<td>SAVE_NVRAM_REGION</td>
<td>145</td>
</tr>
<tr>
<td>USER_D_CACHE_MODE</td>
<td>27</td>
</tr>
<tr>
<td>USER_1_CACHE_MODE</td>
<td>27</td>
</tr>
<tr>
<td>VXB_DEBUG_MSG</td>
<td>313</td>
</tr>
<tr>
<td>VXB_TIMER_DEBUG</td>
<td>313</td>
</tr>
<tr>
<td>VxBus register access</td>
<td>302</td>
</tr>
<tr>
<td>malloc( )</td>
<td>8, 29, 33</td>
</tr>
<tr>
<td>memalign( )</td>
<td>29</td>
</tr>
<tr>
<td>memory allocation</td>
<td></td>
</tr>
<tr>
<td>during VxBus initialization</td>
<td>284</td>
</tr>
<tr>
<td>memory device</td>
<td>277</td>
</tr>
<tr>
<td>memory drivers</td>
<td>266</td>
</tr>
<tr>
<td>memory technology driver</td>
<td></td>
</tr>
<tr>
<td>see MTDs</td>
<td></td>
</tr>
<tr>
<td>memory-mapped systems</td>
<td>9</td>
</tr>
<tr>
<td>method ID</td>
<td>279, 288, 312</td>
</tr>
<tr>
<td>MMU_TAGGING</td>
<td>30, 33</td>
</tr>
<tr>
<td>MTDs</td>
<td>142, 143</td>
</tr>
<tr>
<td>defining as components</td>
<td>155</td>
</tr>
<tr>
<td>defining in the socket driver file</td>
<td>156</td>
</tr>
<tr>
<td>identification routine</td>
<td>148</td>
</tr>
<tr>
<td>initializing the FLLFlash</td>
<td></td>
</tr>
<tr>
<td>structure members</td>
<td>149</td>
</tr>
<tr>
<td>registering an identification routine</td>
<td>156</td>
</tr>
<tr>
<td>writing</td>
<td>148</td>
</tr>
<tr>
<td>writing a map routine</td>
<td>152</td>
</tr>
<tr>
<td>writing read, write, and erase routines</td>
<td>153</td>
</tr>
<tr>
<td>mtdTable[ ]</td>
<td>149, 156</td>
</tr>
<tr>
<td>MULTI_TABLE</td>
<td>116</td>
</tr>
<tr>
<td>multi-function</td>
<td></td>
</tr>
<tr>
<td>device driver naming conventions</td>
<td>311</td>
</tr>
<tr>
<td>processors</td>
<td>10</td>
</tr>
<tr>
<td>multi-mode serial drivers</td>
<td>268</td>
</tr>
<tr>
<td>see also SIO drivers</td>
<td></td>
</tr>
<tr>
<td>multiplexer</td>
<td></td>
</tr>
<tr>
<td>see MUX</td>
<td></td>
</tr>
<tr>
<td>multi-stage initialization sequence</td>
<td></td>
</tr>
<tr>
<td>VxBus</td>
<td>283</td>
</tr>
<tr>
<td>MUX</td>
<td>38</td>
</tr>
<tr>
<td>defined</td>
<td>38</td>
</tr>
<tr>
<td>entry points</td>
<td>40</td>
</tr>
<tr>
<td>MUX API, interactions with</td>
<td>47</td>
</tr>
</tbody>
</table>
MUX_PROTO_OUTPUT 45
MUX_PROTO_PROMISC 45, 46
MUX_PROTO_SNARF 45
muxAddressForm( ) 46, 48
muxAddrResFuncAdd( ) 49
muxAddrResFuncDel( ) 49
muxAddrResFuncGet( ) 49
muxBind( ) 40, 44, 47
muxBufInit( ) 140
muxDataPacketGet( ) 48
muxDevConnect 332
muxDevLoad( ) 40, 47, 54, 60, 69, 332
muxDevStart( ) 47, 54
muxDevStop( ) 48
muxDevUnload( ) 48
muxError( ) 122
muxIoctl( ) 48
muxMCastAddrAdd( ) 48
muxMCastAddrDel( ) 48
muxMCastAddrGet( ) 48
muxPacketAddrGet( ) 48
muxReceive( ) 46, 48, 73
muxSend( ) 46, 48
muxShutdown( ) 48
muxTxRestart( ) 48
muxUnbind( ) 40, 47, 48
muxUnload( ) 333

netPoolCreate( ) 55, 65, 67
netPoolInit( ) 65
netPoolRelease( ) 108
netTupleGet( ) 72

one-shot timer 235
optimization levels in VxBus 281
optimized access macros hEnd 336

PC
pAccess 291, 301, 304
packets, handling
   reception 69
   transmission 82
parent 279
partial snooping 31, 33
pBusSpecificDevInfo 300
PCI
   registration in VxBus 300
   PCI_DEVVEND 300
   PCI_DRIVER_REGISTRATION 300
   PCMCIA socket drivers 158
   pDrvBusFuncs 299
   pDrvCtrl 304
   per-device data structure 15, 24
   per-driver data structure 15, 24
   performance testing 7, 26
   periodic interrupt timer 235
   PLB 276, 283
      configuring the device name 311
      hEnd hwconf.c 330
      initialization routine 276
      registration 300
   PLB_DRIVER_REGISTRATION 300
   pMethods 288
   pNext 299

N
naming
   drivers 13
   macros 17
NAND devices 148
NET_FUNCS 100, 102
NET_PROTOCOL 47, 49
netBufLib 55, 65
netif driver 131
   entry points 132
   porting to the END driver model 139
netJobAdd( ) 56, 58, 74, 133, 137, 336
netJobRing 57, 74, 82
netMblkToBufCopy( ) 113

346
Index

porting
- a driver from an earlier version of VxWorks 21
- a legacy driver to the VxBus model 21
- an END driver from another OS 120
- drivers from another OS 22
preprocessor conditional expressions 16
pRetVal 308
printf( ) 26, 284
probe 279
probe routine 279, 285, 299
probeDatum 308
processing element device 277
processor local bus
  see PLB
project facility
  adding MTD components 155
  CDF entries 21
  CDL 21
  integrating drivers 20
protocol data structure 49

Q
quiescent state 20

R
receive handler interlocking flag 70, 74
regBase 300
register access
  in VxBus 281
  routines 304
    optimized 304
    VxBus 302
  VxBus 291
register probe routine 287, 308
registering
  a driver with VxBus 21
  devices and bus types with VxBus 316
registerRead16( ) 302
registerRead32( ) 302
registerRead64( ) 302
registerRead8( ) 302
registers
  in configuration space 305
registerWrite16( ) 303
registerWrite32( ) 303
registerWrite64( ) 303
registerWrite8( ) 303
registration
  VxBus 298
resident flash array
  see RFA
RFA 158
rfACardDetected( ) 160, 163
rfAAndGetAndClearChangeIndicator( ) 162, 165
rfARegister( ) 159, 162
rfASetMappingContext( ) 162, 165
rfASetWindow( ) 160, 161, 164
rfASocketInit( ) 160, 161, 164
rfAVccOff( ) 160, 163
rfAVccOn( ) 160, 163
rfAVppOff( ) 161, 164
rfAVppOn( ) 160, 164
rfAWriteProtected( ) 162, 165
RFC 1213 123
RFC2233 123
RISC architectures 30
routines
  arpresolve( ) 53
  cache 27
  cacheDmaMalloc( ) 27, 28, 29, 33, 34
  devEndLoad( ) 332
  device creation 15
  endM2Free( ) 125
  endM2Init( ) 124
  endM2Ioctl( ) 125
  endM2Packet( ) 125
  ether_attach( ) 135
  ether_output( ) 133
  etherInputHook( ) 46
  etherOutputHook( ) 46
  flDelayMsec( ) 160
  flDontNeedVpp( ) 153
  flNeedVpp( ) 153
  flWriteProtected( ) 153
  hwMemAlloc( ) 315
intConnect() 15, 26, 292
intLock() 8
jobQueueCreate() 332
jobQueueProcess() 332
logMsg() 25, 26, 335
malloc() 8, 29, 33
memalign() 29
memalign() 335
MTD helper routines 153
muxAddressForm() 46, 48
muxAddrResFuncAdd() 49
muxAddrResFuncDel() 49
muxAddrResFuncGet() 49
muxBind() 40, 44, 47
muxBufInit() 140
muxDataPacketGet() 48
muxDevLoad() 40, 47, 54, 60, 69, 332
muxDevStart() 47, 54
muxDevStop() 48
muxDrvUnload() 48
muxError() 122
muxIoctl() 48
muxMCastAddrAdd() 48
muxMCastAddrDel() 48
muxMCastAddrGet() 48
muxPacketAddrGet() 48
muxPacketDataGet() 48
muxReceive() 46, 48, 73
muxSend() 46, 48
muxShutdown() 48
muxTxRestart() 48
muxUnbind() 40, 47, 48
muxUnLoad() 333
netJobAdd() 56, 58, 74, 133, 137, 336
netPoolCreate() 55, 65, 67
netPoolInit() 65
netPoolRelease() 108
netTupleGet() 72
printf() 26, 284
register access 304
optimized routines
provided by the BSP 304
registerRead16() 302
registerWrite16() 303
registerWrite32() 303
registerWrite64() 303
registerWrite8() 303
rfaRegister() 159
rfaSetWindow() 160
rfaSocketInit() 160
csciCacheSnoopDisable() 205
csciCacheSnoopEnable() 204
csciCacheSynchronize() 205
csciCtrlInit() 194, 203
csIdentMsgBuild() 204
csIdentMsgParse() 204
csciMgrBusReset() 204
csciMgrCtrlEvent() 204
csciMgrEventNotify() 194, 203
csciMgrThreadEvent() 204
csciMsgInComplete() 204
csciMsgOutComplete() 204
csciMsgOutReject() 204
csciSyncXferNegotiate() 204
csciThreadInit() 205
csciTransact() 189
csciWideXferNegotiate() 203
stackError() 50
sysHwInit() 20, 284
sysHwInit2() 12, 284
sysInByte() 120
sysInLong() 120
sysInWord() 120
sysOutByte() 120
sysOutLong() 120
sysOutWord() 120
sysScsiInit() 224
sysTffsInit() 158, 159
sysTimestamp() 257
sysTimestampConnect() 255
sysTimestampDisable() 256
sysTimestampEnable() 256
sysTimestampFreq() 256
sysTimestampLock() 258
sysTimestampPeriod() 256
sysTimestampRoutine() 258
taskLock() 8
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmpltDLParamParse()</td>
<td>337</td>
</tr>
<tr>
<td>tmpltIntConfig()</td>
<td>335</td>
</tr>
<tr>
<td>tmpltRxDescStatusClear()</td>
<td>335</td>
</tr>
<tr>
<td>tmpltRxDescStatusGet()</td>
<td>335</td>
</tr>
<tr>
<td>tmpltTxDescInit()</td>
<td>335</td>
</tr>
<tr>
<td>tmpltTxDescStatusGet()</td>
<td>336</td>
</tr>
<tr>
<td>tmpltTxInt()</td>
<td>336</td>
</tr>
<tr>
<td>tmpltTxPktSend()</td>
<td>335</td>
</tr>
<tr>
<td>usrNetInit()</td>
<td>59</td>
</tr>
<tr>
<td>volRegRead()</td>
<td>306, 307</td>
</tr>
<tr>
<td>volRegWrite()</td>
<td>306, 307</td>
</tr>
<tr>
<td>vxbAddrProbe()</td>
<td>308</td>
</tr>
<tr>
<td>vxbBusAnnounce()</td>
<td>316</td>
</tr>
<tr>
<td>vxbBusPrint()</td>
<td>312</td>
</tr>
<tr>
<td>vxbBusTypeRegister()</td>
<td>316</td>
</tr>
<tr>
<td>vxbCfgRead()</td>
<td>305</td>
</tr>
<tr>
<td>vxbCfgWrite()</td>
<td>305</td>
</tr>
<tr>
<td>vxbDelay()</td>
<td>321</td>
</tr>
<tr>
<td>vxbDevAccessShow()</td>
<td>312</td>
</tr>
<tr>
<td>vxbDeviceAnnounce()</td>
<td>312, 316</td>
</tr>
<tr>
<td>vxbDevIterate()</td>
<td>288, 289, 290</td>
</tr>
<tr>
<td>vxbDevMethodGet()</td>
<td>35, 288, 289, 290</td>
</tr>
<tr>
<td>vxbDevMethodRun()</td>
<td>288, 289, 290</td>
</tr>
<tr>
<td>vxbDevParent()</td>
<td>282</td>
</tr>
<tr>
<td>vxbDevPath()</td>
<td>282</td>
</tr>
<tr>
<td>vxbDevPathShow()</td>
<td>312</td>
</tr>
<tr>
<td>vxbDevRegister()</td>
<td>22</td>
</tr>
<tr>
<td>vxbDevStructShow()</td>
<td>312</td>
</tr>
<tr>
<td>vxbDriverRegister()</td>
<td>288, 299</td>
</tr>
<tr>
<td>vxbInstParamSet()</td>
<td>330, 332</td>
</tr>
<tr>
<td>vxbIntAcknowledge()</td>
<td>293, 295, 309</td>
</tr>
<tr>
<td>vxbIntConnect()</td>
<td>292, 293, 309</td>
</tr>
<tr>
<td>vxbIntDisable()</td>
<td>293, 295, 309</td>
</tr>
<tr>
<td>vxbIntDisconnect()</td>
<td>293, 294, 309</td>
</tr>
<tr>
<td>vxbIntEnable()</td>
<td>293, 295, 309</td>
</tr>
<tr>
<td>vxbMsDelay()</td>
<td>321</td>
</tr>
<tr>
<td>vxbShow()</td>
<td>300</td>
</tr>
<tr>
<td>vxbTimerShow()</td>
<td>312</td>
</tr>
<tr>
<td>vxbTopoShow()</td>
<td>312</td>
</tr>
<tr>
<td>VxBus configuration space access</td>
<td>305</td>
</tr>
<tr>
<td>VxBus register access</td>
<td>302</td>
</tr>
<tr>
<td>VxBus show routines</td>
<td>312</td>
</tr>
<tr>
<td>vxbUsDelay()</td>
<td>321</td>
</tr>
<tr>
<td>vxbUsShow()</td>
<td>312</td>
</tr>
<tr>
<td>vxMemProbe()</td>
<td>287, 301, 308</td>
</tr>
</tbody>
</table>

**S**

SAVE_NVRAM_REGION 145, 146

scatter-gather 82

SCSI

commands 189

controller libraries 192
direct access library 192

module layout 186

objects and data structures 187

sequential access library 193

SCSI drivers 185

advanced controller driver example 205

advanced I/O processor example 218

basic controller example 195

BSP interface 224
data coherency problems 228
development 226

programming interface 193

sysScsi.c template 224
test suites 228

VxWorks interface 190

SCSI manager 190

SCSI_PHYS_DEV 189

SCSI_TRANSACTION 189

scsi2Lib 189

scsi2Lib.h 203

scsiCacheSnoopDisable() 205

scsiCacheSnoopEnable() 204

scsiCacheSynchronize() 205

scsiCommonLib 189, 193

scsiCtrlInit() 194, 203

scsiCtrlLib 192

scsiDirectLib 189, 192

scsiDiskTest() 229

scsiDiskThrputTest() 228

scsiIdentMsgBuild() 204

scsiIdentMsgParse() 204

scsiMgrBusReset() 204
scsiMgrCtrlEvent( ) 204
scsiMgrEventNotify( ) 194, 203
scsiMgrLib 190
scsiMsgInComplete( ) 204
scsiMsgOutComplete( ) 204
scsiMsgOutReject( ) 204
scsiSeqLib 190, 193
scsiSpeedTest( ) 230
scsiSyncXferNegotiate( ) 204
scsiThreadInit( ) 205
scsiTransact() 189
scsiWideXferNegotiate( ) 203
serial drivers 272
  see also multi-mode serial drivers
services
  VxBus 317
setWindow( ) 161
shared memory devices 29
  SHARED_CACHE_LINES 30, 33
  SHARED_POINTERS 31, 34
SIO drivers 268
  see also multi-mode serial drivers
    polled mode 271
SIO_CHAN 269
SIO_DRV_FUNCS 269
sioLib.h 269
SL 323
SNOOPED 31, 33
socket address mapping 165
socket drivers 157
  PCMCIA 158, 162
  RFA 158
socket windowing 165
SOCKET_12_VOLTS 164
stackError( ) 40, 50
stackRcvRtn( ) 40, 49
stackShutdownRtn( ) 40, 51, 52
stackTxRestartRtn( ) 40, 52
standard register device 277
storing your driver 13
structure pointers
  pAccess 301
structures
  DRV_CTRL 315, 328

HEND_RX_QUEUE_PARAM 330, 332
HEND_TX_QUEUE_PARAM 330
PCI_DEVVENDE 300
PCI_DRIVER_REGISTRATION 300
PLB_DRIVER_REGISTRATION 300
TMPLT_DRV_CTRL 335
using 17
VXB_DEVICE 288, 291, 292, 303
VXB_DRIVER 298, 299, 300
vxdbDevRegInfo 311
sysDev.c 20
sysHwInit() 20, 284
sysHwInit2() 12, 284
sysInByte( ) 120
sysInLong( ) 120
sysInstParamTable 330
sysInWord( ) 120
sysLib.c 20
sysOutByte( ) 120
sysOutLong( ) 120
sysOutWord( ) 120
sysPhysMemDesc[ ] 27
sysScsi.c
  sysScsiInit( ) 224
    template 224
sysScsiInit( ) 224
sysSerial.c 20
sysTffs.c 156
sysTffsInit( ) 158, 159
sysTimestamp() 257
sysTimestampConnect( ) 255
sysTimestampDisable() 256
sysTimestampEnable() 256
sysTimestampFreq( ) 256
sysTimestampLock( ) 258
sysTimestampPeriod( ) 256
sysTimestampRoutine( ) 258

tapeFsTest( ) 231
taskLock( ) 8
Index

tasks
  tNetTask  54, 55, 58, 133
  tUsrRoot  54
TCP/IP  37
TCP/IP checksum offloading  88
  checksum support flags  90
  END_CAPABILITIES structure  89, 92
templateSio.c  270
terminology  3
tfsConfig.c  149, 157
third party BSPs, storing drivers  13
timer abstraction routines
  VxBus  321
timers, hardware, characteristics of  236
timestamp drivers  233
  BSP interface  254
  components  239
  configuration  254
  sample drivers  240
  VxWorks interface  238
  VxWorks requirements  237
  working with the System Viewer  238
timestamp timer  235
tmplt.c  334
TMPLT_DRV_CTRL  335
TMPLT_REGISTER_READ32  336
TMPLT_REGISTER_WRITE32  336
tmpltDLInit()  335
tmpltDLParamParse()  337
tmpltHEnd.h  334
tmpltIntConfig()  335
tmpltRxDescStatusClear()  335
tmpltRxDescStatusGet()  335
tmpltTxDescInit()  335
tmpltTxDescStatusGet()  336
tmpltTxInt()  336
tmpltTx_pktSend()  335
tNetTask  54, 55, 58, 133, 331, 336
  token merging  329
  transmit interrupts
    hEnd  336
    transmit-packet-complete handler interlocking flag  82
  TrueFFS  141
    driver development  143
    erase units  178
    flash translation layer (FTL)  166
    socket drivers  157
  TrueFFS layers
    core layer  142
    flash translation layer (FTL)  143
    MTD layer  142
    socket layer  143
  TS_SKEW  260
tuple defined  55
tuple, memory pool  64
tUsrRoot  54

U
unloading
  an END driver  106
  an hEnd driver  333
upstream  280
USER_D_CACHE_MODE  27
USER_DATA_UNKNOWN  30, 33
USER_I_CACHE_MODE  27
usrNetInit()  59

V
VBM  169
Vcc  163
virtual block map  169
virtual buses
  VxBus  310
virtual memory library  27
volatile registers
  VxBus  306
volRegRead()  306, 307
volRegWrite()  306, 307
Vpp  164
VXB_DEBUG_MSG  313
VXB_DEVICE  291, 292, 303
pMethods  288
VxWorks
Device Driver Developer’s Guide, 6.4

VXB_DEVICE_ID 292
VXB_DRIVER 298, 299, 300
   fields 299
VXB_INST_PARAM_OVERRIDE 319, 331
VXB_INST_PARAM_VALUE 319
VXB_ITERATEINSTANCES 290
VXB_ITERATEORPHANS 291
VXB_ITERATEVERBOSE 291
VXB_PARAMETERS 319
VXB_TIMER_DEBUG 313
vxbAddrProbe() 308
vxbBusAnnounce() 316
vxbBusListPrint() 312
vxbBusTypeRegister() 316
vxbCfgRead() 305
vxbCfgWrite() 305
vxbDelay() 321
vxbDevAccessShow() 312
vxbDeviceAnnounce() 312, 316
vxbDevIterate() 288, 289, 290
vxbDevMethodGet() 35, 288, 289, 290
vxbDevMethodRun() 288, 289, 290
vxbDevParent() 282
vxbDevPath() 282
vxbDevPathShow() 312
vxbDevRegInfo 311
vxbDevRegister() 22
vxbDevStructShow() 312
vxbDriverRegister() 298, 299
vxbInstParamSet() 330, 332
vxbIntAcknowledge() 293, 295, 309
vxbIntConnect() 292, 293, 309
vxbIntDisable() 293, 295, 309
vxbIntDisconnect() 293, 294, 309
vxbIntEnable() 293, 295, 309
vxbMsDelay() 321
vxbShow() 300
vxbTimerShow() 312
vxbTopoShow() 312
VxBus
   about 275
   accessing registers through 22
   bus controller device drivers 12, 315
   bus controller devices 280
   bus support 11
   bus topology API 282
   configuration space access routines 305
   configuration space registers
      reading 306
      writing to 306
   creating an instance 284, 298
   debug messages 313
   debug registers 22
   defining new interfaces 282
   device discovery 280
   device driver initialization sequence 298
   device driver interrupts 308
   device driver registration 298
   device drivers
      common issues 311
      debugging 312
   device enumeration 316
   device initialization
      handling incompatible device variants 286
   topology issues 286
   device insertion and removal 281, 282
   device models 277
      memory device 277
      processing element device 277
      standard register device 277
   driver and device interface 281
   driver and OS interface 281
   driver and VxBus interface 282
   driver methods 281, 282, 288
      classes, types, and macros 290
      OS and middleware access 288
   hardware access abstraction 328
   hEnd 325
   initialization
      entry points 299
   initialization process
      first stage 284
      OS and middleware 285
      second stage 284
      third stage 285
   initialization sequence 298
   integrating
      driver code with 313
   interrupt management 292
interrupt sources 309
interrupts 292
method ID 288, 312
modifying BSP initialization 22
multi-function device drivers 310
non-optimized register access 22
optimization levels for device register access 281
parameter system 317
data structures 319
PCI registration 300
PLB registration 300
porting a legacy driver to 21
probe routine 284, 285, 299
register access 291
  optimized routine macros 304
register access macros 302
register access routines 302
register probe routine 287, 308
registering
  devices and bus types 316
registering a driver with 21
registers in configuration space 305
registration 298
services 280, 317
  example methods and utilities 317
show routines 312
standard device registers 302
terms
  access routine 278
  bus 278
  bus controller 278
  bus discovery 278
  bus match 278
  bus type 278
  child 278
  device 278
downstream 278
  driver 279
driver method 279
instance 279
method ID 279
parent 279
probe 279
probe routine 279
upstream 280
timer abstraction routines 321
virtual buses 310
volatile registers 306
VxBus initialization sequence 282
vxBusDelay( ) 321
vxBusShow( ) 312
vxMemProbe( ) 287, 301, 308
W
WDB agent 45
WDB_COMM_END 45
wdDelete( ) 108
Wind River System Viewer 233
write pipelining 28, 30, 32
WRITE_PIPING 28, 30, 32
writing
  a multi-function device driver 310
  a new driver using the legacy model 23
  a new END driver 119
  a new hEnd driver 333
  DL header files 334
  DL source files 334
driver documentation 13
MTD components 148
wvTmrRegister( ) 239