WIND RIVER

# VxWorks®

# DEVICE DRIVER DEVELOPER'S GUIDE
## Volume 2: Writing Class-Specific Device Drivers

6.6

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

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

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

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

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

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



13 Nov 07
Part #: DOC-16099-ND-00

# Contents


1 Class-Specific Driver Development ..... 1

1.1 About VxBus Driver Classes ..... 1

1.2 Before You Begin ..... 2

1.3 About this Document ..... 2

Navigating this Documentation Set ..... 2

2 Bus Controller Drivers ..... 3

2.1 Introduction ..... 3

2.2 Overview ..... 4

2.3 VxBus Driver Methods ..... 5

2.3.1 {busCtlrCfgRead}( ) ..... 6

2.3.2 {busCtlrCfgWrite}( ) ..... 7

2.3.3 {busCtlrAccessOverride}( ) ..... 8

Override for (*busCfgRead)( ) ..... 9
Override for (*busCfgWrite)( ) ..... 9
Override for (*vxbDevControl)( ) ..... 9

2.3.4 {busCtlrCfgInfo}( ) ..... 10

2.3.5 {busCtlrBaseAddrCvt}( ) ..... 11

2.3.6 {vxbDevRegMap}( ) ..... 11
Specifying a Predefined Transaction Type ..... 13
Providing a New Transaction Type ..... 15
2.3.7 {vxbIntDynaVecProgram}( ) ..... 17

**2.4 Header Files ..... 17**

**2.5 BSP Configuration ..... 17**

2.5.1 PCI Configuration ..... 18
2.5.2 PCI Autoconfiguration ..... 19

**2.6 Available Utility Routines ..... 20**

2.6.1 PCI Configuration ..... 20
2.6.2 PCI Autoconfiguration ..... 21
2.6.3 vxbBusAnnounce( ) ..... 21
2.6.4 vxbPciBusTypeInit( ) ..... 22
2.6.5 vxbPciDeviceAnnounce( ) ..... 22

**2.7 Initialization ..... 23**

2.7.1 Initialization Example ..... 24
vxbBusAnnounce( ) ..... 25
vxbDeviceAnnounce( ) ..... 26
vxbDevStructAlloc( ) ..... 26
vxbDevStructFree( ) ..... 26

**2.8 Debugging ..... 26**

**3 Direct Memory Access Drivers ..... 29**

**3.1 Introduction ..... 29**

**3.2 Overview ..... 30**

**3.3 VxBus Driver Methods ..... 30**

3.3.1 {vxbDmaResourceGet}( ) ..... 30
3.3.2 {vxbDmaResourceRelease}( ) ..... 31

3.3.3 {vxbDmaResDedicatedGet}( ) ..... 31

**3.4 Header Files ..... 32**

**3.5 BSP Configuration ..... 32**

**3.6 Available Utility Routines ..... 33**

**3.7 Initialization ..... 33**

**3.8 DMA System Structures and Routines ..... 33**

3.8.1 (*dmaRead)( ) ..... 34

3.8.2 (*dmaReadAndWait)( ) ..... 34

3.8.3 (*dmaWrite)( ) ..... 34

3.8.4 (*dmaWriteAndWait)( ) ..... 35

3.8.5 (*dmaCancel)( ) ..... 35

3.8.6 (*dmaPause)( ) ..... 35

3.8.7 (*dmaResume)( ) ..... 36

3.8.8 (*dmaStatus)( ) ..... 36

**3.9 Debugging ..... 36**

**4 Interrupt Controller Drivers ..... 37**

**4.1 Introduction ..... 38**

**4.2 Overview ..... 38**

Interrupt Identification ..... 39
Interrupt Controller Driver Responsibilities ..... 39
Interrupt Controller Configurations ..... 39
Dynamic Vectors ..... 40
Interrupt Controller Drivers and Multiprocessing ..... 40

**4.3 VxBus Driver Methods ..... 41**

4.3.1 Basic Methods ..... 41
{vxbIntCtlrConnect}( ) ..... 41
{vxbIntCtlrDisconnect}( ) ..... 41
{vxbIntCtlrEnable}( ) ..... 42
{vxbIntCtlrDisable}( ) ..... 42
4.3.2 Dynamic Vector Methods ..... 43
{vxbIntDynaVecConnect}( ) ..... 43
4.3.3 Multiprocessor Methods ..... 43
{vxbIntCtlrIntReroute}( ) ..... 43
{vxbIntCtlrCpuReroute}( ) ..... 44
{vxIpiControlGet}( ) ..... 44
**4.4 Header Files ..... 45**
vxbIntrCtlr.h ..... 45
vxbIntCtlrLib.h ..... 45
**4.5 BSP Configuration ..... 45**
4.5.1 Interrupt Input Table ..... 46
4.5.2 Dynamic Vector Table ..... 48
4.5.3 CPU Routing Table ..... 49
4.5.4 Interrupt Priority ..... 50
4.5.5 Crossbar Routing Table ..... 50
**4.6 Available Utility Routines ..... 51**
4.6.1 intCtlrHwConfGet( ) ..... 52
4.6.2 intCtlrISRAdd( ) ..... 52
4.6.3 intCtlrISRDisable( ) ..... 52
4.6.4 intCtlrISREnable( ) ..... 52
4.6.5 intCtlrISRRemove( ) ..... 53
4.6.6 intCtlrPinFind( ) ..... 53
4.6.7 intCtlrTableArgGet( ) ..... 53
4.6.8 intCtlrTableFlagsGet( ) ..... 53
4.6.9 intCtlrTableIsrGet( ) ..... 53

4.6.10 intCtlrHwConfShow( ) ..... 53
4.6.11 intCtlrTableCreate( ) ..... 54
4.6.12 intCtlrTableFlagsSet( ) ..... 54
4.6.13 intCtlrTableUserSet( ) ..... 54
4.6.14 VXB_INTCTLR_ISR_CALL( ) ..... 54
4.6.15 VXB_INTCTLR_PINENTRY_ENABLED( ) ..... 55
4.6.16 VXB_INTCTLR_PINENTRY_ALLOCATED( ) ..... 55
4.6.17 Dispatch Routines ..... 55
vxbIntDynaCtlrInputInit( ) ..... 56
vxbIntDynaVecProgram( ) ..... 56

**4.7 Initialization ..... 56**

**4.8 Interrupt Controller Typologies and Hierarchies ..... 57**

**4.9 Interrupt Priority ..... 58**

**4.10 ISR Dispatch ..... 59**

**4.11 Managing Dynamic Interrupt Vectors ..... 62**

Configuring Dynamic Vectors Using the Service Driver Routines ... 63
Configuring Dynamic Vectors in the BSP ..... 63
Programming Dynamic Vectors ..... 64
Determining Dynamic Vector Values ..... 65

**4.12 Internal Representation of Interrupt Inputs ..... 65**

**4.13 Multiprocessor Issues with VxWorks SMP ..... 66**

4.13.1 Routing Interrupt Inputs to Individual CPUs ..... 67
4.13.2 Interprocessor Interrupts ..... 68
4.13.3 Limitations in Multiprocessor Systems ..... 72

**4.14 Debugging ..... 73**

5 Multifunction Drivers ..... 75

5.1 Introduction ..... 75

5.2 Overview ..... 76

5.3 VxBus Driver Methods ..... 76

5.4 Header Files ..... 77

5.5 BSP Configuration ..... 77

5.6 Available Utility Routines ..... 78

vxbDevStructAlloc( ) ..... 78
vxbDeviceAnnounce( ) ..... 78
vxbDevRemovalAnnounce( ) ..... 78
vxbDevStructFree( ) ..... 78
vxbBusAnnounce( ) ..... 79

5.7 Initialization ..... 79

5.8 Device Interconnections ..... 79

5.8.1 Interleaved Registers ..... 79

5.8.2 Shared Resources ..... 81

5.8.3 Other Interactions ..... 81

5.9 Logical Location of Subordinate Devices ..... 81

5.10 Debugging ..... 82

6 Network Drivers ..... 83

6.1 Introduction ..... 83

6.1.1 Terminology ..... 84

6.1.2 Networking Overview ..... 84

Seven Layer OSI Model ..... 84
Transmission Media and VxWorks ..... 85
Protocols ..... 85

**6.2 Network Interface Drivers ..... 86**

6.2.1 Network Interface Driver Overview ..... 86

Functional Modules ..... 86
Network Driver Interrupts ..... 88

6.2.2 VxBus Driver Methods for Network Interface Drivers ..... 88

{muxDevConnect}( ) ..... 89
{vxbDrvUnlink}( ) ..... 90
{miiMediaUpdate}( ) ..... 90
{miiRead}( ) ..... 90
{miiWrite}( ) ..... 91

6.2.3 Header Files for Network Interface Drivers ..... 92

6.2.4 BSP Configuration for Network Interface Drivers ..... 93

6.2.5 Available Utility Routines for Network Interface Drivers ..... 94

MUX Interactions ..... 95
Job Queueing ..... 96
Buffer Management ..... 97
DMA Support ..... 98
PHY and MII bus interactions ..... 99

6.2.6 Initialization for Network Interface Drivers ..... 101

6.2.7 MUX: Connecting to Networking Code ..... 102

6.2.8 jobQueueLib: Deferring ISRs ..... 103

6.2.9 netBufLib: Transferring Data with mBlks ..... 104

Setting Up a Memory Pool ..... 104
Supporting Scatter-Gather ..... 105

6.2.10 vxbDmaBufLib: Managing DMA ..... 106

6.2.11 Protocol Impact on Drivers ..... 107

IPv4 and IPv6 Checksum Offloading ..... 107

6.2.12 Other Network Interface Driver Issues ..... 115

Receive Handler Interlocking Flag ..... 115
Fair Received Packet Handling ..... 116
Receive Stall Handling ..... 117

6.2.13 Debugging Network Interface Drivers ..... 118

Using VxBus Show Routines ..... 118
Deferring Driver Registration ..... 118
Pairing with a PHY instance ..... 119
Stress Testing ..... 120
Netperf Test Suite ..... 120
Interrupt Validation ..... 120
Additional Tests ..... 120

**6.3 PHY Drivers ..... 123**

6.3.1 PHY Driver Overview ..... 123

PHY Device Probing and Discovery ..... 124
MAC and MII Bus Relationship ..... 125
Generic PHY Driver Support ..... 125
Generic TBI Driver Support ..... 126

6.3.2 VxBus Driver Methods for PHY Drivers ..... 126

Upper Edge Methods ..... 127
Lower Edge Methods ..... 127

6.3.3 Header Files for PHY Drivers ..... 130

6.3.4 BSP Configuration for PHY Drivers ..... 130

6.3.5 Available Utility Routines for PHY Drivers ..... 130

Upper Edge Utility Routines ..... 130
Lower Edge Utility Routines ..... 131

6.3.6 Initialization for PHY Drivers ..... 132

6.3.7 Debugging PHY Drivers ..... 133

**6.4 Wireless Ethernet Drivers ..... 133**

**6.5 Hierarchical END Drivers ..... 134**

**7 Non-Volatile RAM Drivers ..... 135**

**7.1 Introduction ..... 135**

NVRAM Drivers and TrueFFS ..... 135

**7.2 Non-Volatile RAM Drivers ..... 136**

7.2.1 NVRAM Driver Overview ..... 136
7.2.2 VxBus Driver Methods for NVRAM Drivers ..... 137
{nonVolGet}( ) ..... 137
{nonVolSet}( ) ..... 137
7.2.3 Header Files ..... 137
7.2.4 BSP Configuration for NVRAM Drivers ..... 138
7.2.5 Utility Routines for NVRAM Drivers ..... 139
7.2.6 Initialization for NVRAM Drivers ..... 139
7.2.7 NVRAM Block Sizes ..... 139
7.2.8 Stacking NVRAM Instances ..... 140
7.2.9 Debugging NVRAM Drivers ..... 141

**7.3 Flash File System Support with TrueFFS ..... 141**

7.3.1 TrueFFS Overview ..... 141
Core Layer ..... 142
MTD Layer ..... 142
Socket Layer ..... 142
Flash Translation Layer ..... 143
7.3.2 TrueFFS Driver Development Process ..... 143
Using MTD-Supported Flash Devices ..... 143
Writing MTD Components ..... 148
Socket Drivers ..... 157
Flash Translation Layer ..... 166

**8 Resource Drivers ..... 185**

**8.1 Introduction ..... 185**

**8.2 Overview ..... 186**

**8.3 VxBus Driver Methods ..... 186**

**8.4 Header Files ..... 187**

**8.5 BSP Configuration ..... 187**

8.6 Available Utility Routines ..... 187
8.7 Initialization ..... 187
8.8 Debugging ..... 188

9 Serial Drivers ..... 189

9.1 Introduction ..... 190
9.2 Overview ..... 190
9.3 VxBus Driver Methods ..... 190
9.3.1 {sioChanGet}( ) ..... 191
9.3.2 {sioChanConnect}( ) ..... 191
9.4 Header Files ..... 192
9.5 BSP Configuration ..... 193
9.6 Available Utility Routines ..... 193
9.7 Initialization ..... 193
9.8 Polled Mode Versus Interrupt-Driven Mode ..... 194
9.9 SIO_CHAN and SIO_DRV_FUNCS ..... 194
9.10 WDB ..... 197
9.10.1 WDB and Kernel Initialization ..... 197
9.11 Serial Drivers, Initialization, and Interrupts ..... 198
9.11.1 WDB and Interrupts ..... 198
9.11.2 Initialization Order and Interrupts ..... 199
9.11.3 Initialization Order ..... 199
9.12 Debugging ..... 199

**10 Storage Drivers ..... 201**

**10.1 Introduction ..... 201**

**10.2 Overview ..... 202**

**10.3 VxBus Driver Methods ..... 202**

**10.4 Header Files ..... 202**

**10.5 BSP Configuration ..... 203**

**10.6 Available Utility Routines ..... 203**

erfHandlerRegister( ) and erfHandlerUnregister( ) ..... 203
erfEventRaise( ) ..... 203
xbdAttach( ) ..... 204
bio_done( ) ..... 204

**10.7 Initialization ..... 204**

**10.8 Interface with VxWorks File Systems ..... 205**

10.8.1 Device Creation ..... 205

ERF Registration ..... 205
Advertisement of XBD Methods ..... 206
ERF New Device Notification ..... 207

10.8.2 Processing ..... 208

10.8.3 Event Reporting ..... 208

**10.9 Writing New Storage Drivers ..... 210**

**10.10 Debugging ..... 211**

**11 Timer Drivers ..... 213**

**11.1 Introduction ..... 214**

**11.2 Overview ..... 214**

**11.3 VxBus Driver Methods ..... 215**

**11.4 Header Files ..... 218**

**11.5 BSP Configuration ..... 218**

**11.6 Available Utility Routines ..... 219**

**11.7 Initialization ..... 219**

**11.8 Data Structure Layout ..... 219**

**11.9 Implementing Driver Service Routines ..... 221**

11.9.1 (*timerAllocate)( ) ..... 221

11.9.2 (*timerRelease)( ) ..... 222

11.9.3 (*timerRolloverGet)( ) ..... 222

11.9.4 (*timerCountGet)( ) ..... 223

11.9.5 (*timerDisable)( ) ..... 224

11.9.6 (*timerEnable)( ) ..... 224

11.9.7 (*timerISRSet)( ) ..... 225

11.9.8 (*timerEnable64)( ) ..... 226

11.9.9 (*timerRolloverGet64)( ) ..... 226

11.9.10 (*timerCountGet64)( ) ..... 227

**11.10 Integrating a Timer Driver ..... 228**

11.10.1 VxWorks System Clock ..... 228

11.10.2 VxWorks Auxiliary Clock ..... 230

11.10.3 VxWorks Timestamp Driver ..... 231

**11.11 Debugging ..... 232**

**11.12 SMP Considerations ..... 233**

**12 USB Drivers ..... 235**

**12.1 Introduction ..... 235**

**12.2 Wind River USB Overview ..... 236**

12.2.1 USB Host Stack Drivers ..... 236
VxBus Model Drivers ..... 236
Other Host Drivers ..... 236
12.2.2 USB Peripheral Stack Drivers ..... 237

**12.3 Host Controller and Root Hub Class Drivers ..... 237**

12.3.1 VxBus Driver Methods ..... 238
12.3.2 Header Files ..... 238
12.3.3 BSP Configuration ..... 238
12.3.4 Available Utility Routines ..... 241
12.3.5 Initialization ..... 241
12.3.6 Debugging ..... 242

**13 Other Driver Classes ..... 245**

**13.1 Introduction ..... 245**

**13.2 Overview ..... 246**

**13.3 VxBus Driver Methods ..... 246**

**13.4 Header Files ..... 247**

**13.5 BSP Configuration ..... 247**

**13.6 Available Utility Routines ..... 248**

**13.7 Initialization ..... 248**

**13.8 Debugging ..... 248**

**A Glossary ..... 249**

**Index ..... 253**

# 1 Class-Specific Driver Development


1.1 About VxBus Driver Classes 1

1.2 Before You Begin 2

1.3 About this Document 2


## 1.1 About VxBus Driver Classes

As explained in *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*, devices, and the drivers that manage them, can be divided into categories or *classes* based on the particular function the device and driver are expected to perform.

For example, even though they are very different devices, a simple VGA controller (typical of older PCs) and a modern display controller running on PCI Express provide the same functionality in a system. That is, both devices are responsible for displaying graphical information on a video device. This makes both of these devices, as well as the drivers that control them, part of the same class.

## 1.2 Before You Begin

This document assumes you are familiar with the concepts presented in *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*. If you are not an experienced VxWorks device driver developer or you do not have experience with the VxBus driver model, you must review the information provided in Volume 1 before using this document.

If you are migrating or maintaining VxWorks device drivers based on the legacy device driver model, see *VxWorks Device Driver Developer's Guide, Volume 3: Legacy Drivers and Migration*. This volume does not apply to legacy model device drivers.

## 1.3 About this Document

This document provides information on the driver requirements for specific VxBus driver classes (see *1.1 About VxBus Driver Classes*, p.1). The information presented in this document is intended to supplement the information provided in *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*.

**NOTE:** Concepts and procedures presented in Volume 1 apply to class-specific VxBus model device drivers in general, regardless of class.

### Navigating this Documentation Set

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

# 2 Bus Controller Drivers


2.1 Introduction 3

2.2 Overview 4

2.3 VxBus Driver Methods 5

2.4 Header Files 17

2.5 BSP Configuration 17

2.6 Available Utility Routines 20

2.7 Initialization 23

2.8 Debugging 26


## 2.1 Introduction

This chapter describes VxBus bus controller drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 2.2 Overview

Bus controller drivers provide the support services that are required to enable other drivers attached to downstream devices to communicate with their associated hardware in a uniform way. Functionally, a bus controller driver acts as an abstraction layer between a device driver and the hardware that it controls, ensuring that the I/O operations that need to occur between the driver and its device occur correctly on the target hardware.

In addition, a bus controller driver provides support services to VxBus, allowing VxBus to configure the bus for proper operation, discover devices on the bus, and perform other operations that are outside the scope of normal communication that occurs between a driver and its target hardware.

Graphically, a bus controller can be viewed as an interconnect between a driver and its target hardware, and another interconnect between VxBus and the bus being controlled. Figure 2-1 illustrates this communication.

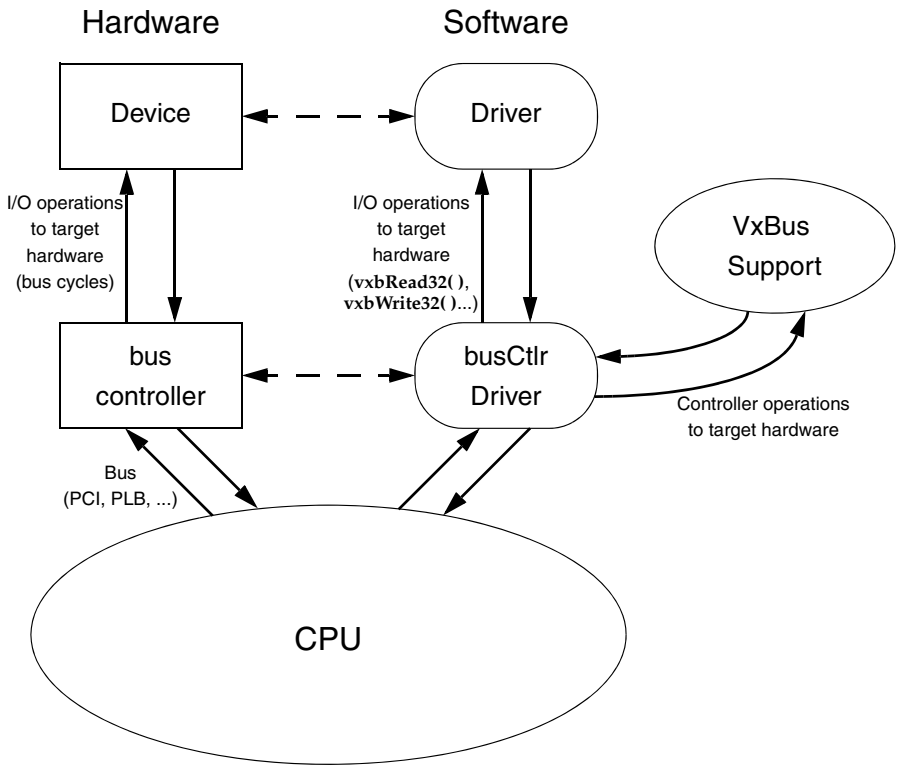
All systems must have at least one bus controller driver. This is because even devices that are directly connected to a CPU must have a parent bus controller. The top-level bus controller is referred to as the processor local bus (PLB) bus controller.

From the PLB, subordinate bus controllers are often available to connect the CPU to devices that are not local to the CPU itself. An example of this is a PCI bus controller located on the PLB bus, which serves as a bridge between the PLB hardware and the PCI bus hardware. The PCI bus controller driver allows the CPU to access the downstream device.

Within VxBus, bus controller drivers are treated like standard drivers in most ways. However, there is a fundamental difference between bus controller drivers and standard drivers; standard drivers typically provide a service to the operating system or to middleware, while bus controller drivers provide a service to other drivers.

Bus controller drivers are relatively complex when compared with other drivers within VxWorks. While the text in this chapter provides information that is necessary in order to successfully develop a driver, you should also refer to the existing bus controller drivers, located in *installDir*/**vxworks-6.***x*/**target/src/hwif/busCtlr**, to see actual implementations and to understand how bus controller drivers interact with VxWorks.

Figure 2-1 **Bus Controller Communication**




## 2.3 **VxBus Driver Methods**

Bus controller drivers use a variety of different driver methods, depending on the type of bus that is being controlled by the driver. In this release, the majority of methods are designed for use with either the PLB or PCI bus types. Each driver method listed in this section lists the bus types that the method is designed to support. Each section also notes if support for the given driver method is optional.

### 2.3.1 {busCtlrCfgRead}( )

The **{busCtlrCfgRead}( )** method is used to read 8, 16, or 32-byte quantities from the configuration space of the bus. Currently, this method is used exclusively on the PCI bus by the PCI bus support code.

Within a bus controller driver, the **{busCtlrCfgRead}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{busCtlrCfgRead}
    (
    VXB_DEVICE_ID   pInst,          /* device info */
    int             bus,            /* bus number */
    int             dev,            /* device number */
    int             func,           /* function number */
    UINT32          byteOffset,     /* offset into config space */
    UINT32          transactionSize,/* transaction size, in bytes */
    char *          pDstBuf,        /* buffer to write to */
    UINT32 *        pFlags          /* flags */
    )
```

The parameters to **func{busCtlrCfgRead}( )** are:

**pInst**
The **VXB_DEVICE_ID** for the bus controller instance.

**bus**
The PCI bus number of the target hardware

**dev**
The PCI device number of the target hardware

**func**
The PCI function number of the target hardware

**byteOffset**
The offset into the configuration space where the read is performed. Because non-aligned configuration accesses are not allowed, **byteOffset** must be an even multiple of the transaction size.

**transactionSize**
The data size to read, in bits. Valid values are 8, 16, and 32.

**pDstBuf**
A pointer to the buffer used to store the value read from configuration space.

**pFlags**
Reserved.

The **func{busCtlrCfgRead}( )** routine performs whatever device-specific operations are required to perform the requested read operation from the bus configuration space.



### 2.3.2 {busCtlrCfgWrite}( )

The **{busCtlrCfgWrite}( )** method is used to write 8, 16, or 32-byte quantities to the configuration space of the bus. Currently, this method is used exclusively on the PCI bus by the PCI bus support code.

Within a bus controller driver, the **{busCtlrCfgWrite}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{busCtlrCfgWrite}
    (
    VXB_DEVICE_ID   pInst,          /* device info */
    int             bus,            /* bus number */
    int             dev,            /* device number */
    int             func,           /* function number */
    UINT32          byteOffset,     /* offset into config space */
    UINT32          transactionSize,/* transaction size, in bytes */
    char *          pSrcBuf,        /* buffer to read from */
    UINT32 *        pFlags          /* flags */
    )
```

The parameters to **func{busCtlrCfgWrite}( )** are:

**pInst**
The **VXB_DEVICE_ID** for the bus controller instance.

**bus**
The PCI bus number of the target hardware

**dev**
The PCI device number of the target hardware

**func**
The PCI function number of the target hardware

**byteOffset**
The offset into the configuration space where the write is performed. Because non-aligned configuration accesses are not allowed, **byteOffset** must be an even multiple of the transaction size.

**transactionSize**
The data size to write, in bits. Valid values are 8, 16, and 32.

**pSrcBuf**
A pointer to the data that will be written to configuration space.

**pFlags**
Reserved.

The **func{busCtlrCfgWrite}( )** routine performs whatever device-specific operations are required to perform the requested write operation to the bus configuration space.

### 2.3.3 {busCtlrAccessOverride}( )

When a bus controller is installed into VxWorks, some service routines are automatically associated with the bus controller, and are made available to device drivers for devices that reside on the bus. These default service routines are not always appropriate for the installed bus. Therefore, you may wish to provide alternate implementations of the services in your bus controller driver. The **{busCtlrAccessOverride}( )** method provides your bus controller driver with a means of overriding selected service routines.

Within a bus controller driver, the **{busCtlrAccessOverride}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS  func{busCtlrAccessOverride}()
    (
    VXB_DEVICE_ID     pInst,    /* device info */
    VXB_ACCESS_LIST * pAccess   /* access structure pointer */
    )
```

A pointer to the **VXB_ACCESS_LIST** data structure is passed to the method. This allows the bus controller driver to replace any of the function pointers that are contained within this data structure with alternate implementations. **VXB_ACCESS_LIST** is declared in *installDir*/**vxworks-6.***x*/**target/src/hwif/h/vxbus/vxbAccess.h**.

Although **VXB_ACCESS_LIST** contains a large collection of function pointers, only three of the function pointers should be modified by the bus controller driver. These pointers are **(*busCfgRead)( )**, **(*busCfgWrite)( )**, and **(*vxbDevControl)( )**. Other fields are considered reserved fields and must be left unchanged by this method. The remainder of this section discusses the fields that can be overridden, reserved fields are not discussed.

#### Override for (*busCfgRead)( )

The prototype for **(*busCfgRead)( )** is:

```
STATUS (*busCfgRead)
    (
    VXB_DEVICE_ID devID,          /* device info */
    UINT32        byteOffset,     /* offset into config space */
    UINT32        transactionSize,/* transaction size, in bytes */
    char *        pDataBuf,       /* buffer to write to */
    UINT32 *      pFlags          /* flags */
    );
```

This routine reads from the bus configuration space. It is used by drivers for devices that reside directly on the bus that is being controlled by the bus controller driver. The [*bus*, *device*, *function*] tuple is not provided directly. Instead, this tuple must be extracted by de-referencing the instance-specific data available using **devID**.

#### Override for (*busCfgWrite)( )

The prototype for **(*busCfgWrite)( )** is:

```
STATUS (*busCfgWrite)
    (
    VXB_DEVICE_ID devID,          /* device info */
    UINT32        byteOffset,     /* offset into config space */
    UINT32        transactionSize,/* transaction size, in bytes */
    char *        pDataBuf,       /* buffer to read from */
    UINT32 *      pFlags          /* flags */
    );
```

This routine writes to the bus configuration space. It is used by drivers for devices that reside directly on the bus that is being controlled by the bus controller driver. The [*bus*, *device*, *function*] tuple is not provided directly. Instead, this tuple must be extracted by de-referencing the instance-specific data available using **devID**.

#### Override for (*vxbDevControl)( )

This routine provides a service similar to an **ioctl( )**, allowing specialized control requests to be delivered to a bus controller driver. In previous releases of VxBus, this routine is used for interrupt management and for device register access. With VxBus version 3, these functions are provided by other modules. Because of this, the **(*vxbDevControl)( )** routine is no longer required, provided that VxBus interrupt controller drivers are used to manage interrupts.

The prototype for **(*vxbDevControl)( )** is:

```
STATUS (*vxbDevControl)
    (
    VXB_DEVICE_ID   devID,          /* device info */
    pVXB_DEVCTL_HDR pBusDevControl  /* parameter */
    );
```

VxBus version 3 simplified the method used for register access. Because of this change, you do not need to provide support for register access routines in your bus controller driver unless some part of bus controller driver code needs to be executed in order to perform the register operation. For more information on this register access, see *2.3.6 {vxbDevRegMap}( )*, p.11.

In past releases, the **(*vxbDevControl)( )** routine was used to allow a bus controller driver to support the configuration of interrupts for devices on the bus being controlled. This service was in keeping with the design goals for bus controller drivers for several releases of VxWorks. However, in this release, bus controller drivers are not responsible for interrupt management for their subordinate devices. Instead, the responsibility for this type of operation has been migrated to interrupt controller device drivers (see *4. Interrupt Controller Drivers*).

### 2.3.4 {busCtlrCfgInfo}( )

VxBus provides a utility library that bus controller drivers can use to support the generation of configuration transactions on the target bus. The utility library makes use of an instance-specific data structure to accomplish its operations. The **{busCtlrCfgInfo}( )** method provides a way for your bus controller driver to export a pointer to this data structure so that the utility library can make use of it.

Within a bus controller driver, the **{busCtlrCfgInfo}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS {busCtlrCfgInfo}()
    (
    VXB_DEVICE_ID pInst,  /* device info */
    char *        pArgs   /* buffer to write to */
    )
```

The implementation of **{busCtlrCfgInfo}( )** is straightforward. The bus controller driver simply returns a pointer to a bus-type specific information structure. For example, see the following PCI code:

```
*(struct vxbPciConfig *) pArgs = pInst->pDrvCtrl->pPciConfig;
```

In this example, the bus controller driver has already allocated the **vxbPciConfig** data structure and stored a pointer to it in its **pDrvCtrl** data area. (For further

details about the use of the **vxbPciConfig** data structure, see *2.6.1 PCI Configuration*, p.20.)



### 2.3.5 {busCtlrBaseAddrCvt}( )

The **{busCtlrBaseAddrCvt}( )** method gives a bus controller driver the opportunity to modify the address of a bus transaction to account for address space differences that happen through the bus controller. At present, only PCI bus controllers use this service.

Within a bus controller driver, the **{busCtlrBaseAddrCvt}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{busCtlrBaseAddrCvt}
    (
    VXB_DEVICE_ID   devID,      /* device info */
    UINT32 *        pBaseAddr   /* pointer to base address */
    )
```

The PCI bus is a memory-mapped bus, with the bus controller acting as an arbiter to forward memory transactions from the originating CPU across the bus so that they are delivered to the target hardware on the bus. When this forwarding occurs, it is common for some type of address translation to take place as the requested transaction crosses the PCI bus controller. The **{busCtlrBaseAddrCvt}( )** method gives the PCI bus controller driver the ability to describe the address translation that takes place.

The **pBaseAddr** pointer that is passed to the method is both an input and an output parameter. On input, it contains a value from one of the base address registers (BARs) of a device on the PCI bus. The driver should modify this value so that it contains a pointer that, when de-referenced, properly points to the location in the CPU address space where the target device is mapped.

### 2.3.6 {vxbDevRegMap}( )

Device drivers use a standard set of routines to read and write to device registers. (This is described in *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.) The standard routines are:

- **vxbRead8( )**
- **vxbRead16( )**
- **vxbRead32( )**
- **vxbRead64( )**

- **vxbWrite8( )**
- **vxbWrite16( )**
- **vxbWrite32( )**
- **vxbWrite64( )**

**NOTE:** In the remainder of this section, these routines as referred to as **vxbRead*( )** and **vxbWrite*( )**.

Unless an alternate implementation is explicitly specified by the bus controller driver, a default implementation for each of these routines is used.

The **{vxbDevRegMap}( )** method is used by a bus controller driver when the driver needs to override the implementation for the various **vxbRead*( )** and **vxbWrite*( )** register access routines. Within a bus controller driver, the **{vxbDevRegMap}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{vxbDevRegMap){}
    (
    VXB_DEVICE_ID pInst,    /* bus controller instance */
    VXB_DEVICE_ID pChild,   /* instance for child of this controller */
    int index,              /* index into pChild->regBase[] */
    void ** pHandle         /* buffer to store handle */
    )
```

For each supported processor architecture family, the **vxbRead*( )** and **vxbWrite*( )** routines support six predefined transaction types as follows:

- memory mapped access (no ordering enforced)
- memory mapped access (ordering enforced)
- I/O space access (ordering implied)
- byte swapped memory mapped access (no ordering enforced)
- byte swapped memory mapped access (ordering enforced)
- byte swapped I/O space access (ordering implied)

When a device driver maps in a portion of its address space by calling **vxbRegMap( )**, the routine checks to see if the parent bus controller associated with the driver instance supports the **{vxbDevRegMap}( )** method. If the bus controller supports this method, **vxbRegMap( )** invokes the bus controller method. If this method is not provided by the bus controller driver, **vxbRegMap( )** provides a reasonable default implementation for the access routine. The default implementation that is chosen is dependent on the target architecture.

The bus controller method **{vxbDevRegMap}( )** is responsible for creating a *handle* to describe the type of transaction to be performed when the driver makes subsequent calls to any of the **vxbRead*( )** or **vxbWrite*( )** routines.

There are two scenarios that the bus controller driver must deal with when implementing this method:

- One of the six predefined transaction types can be used. In this case, the bus controller driver only specifies which transaction type to employ.
- None of the six predefined transaction types can be used. In this case, the bus controller driver provides its own implementation of the service.

Regardless of the scenario, the bus controller is responsible for creating a handle that accurately describes the transaction type that is required to support the underlying hardware.

The handle value that is provided to the **vxbRead*( )** and **vxbWrite*( )** routines is used as an opaque value by the individual device drivers, but a bus controller driver must understand exactly how the handle is used to control the type of transaction that is performed using the **vxbRead*( )** or **vxbWrite*( )** routines. In a typical situation, the handle is treated as a **void *** data type. Because VxWorks is a 32-bit operating system, the handle is encoded using 32-bits. If the 32 bits of the handle are cast to a **UINT32** data type, the integer value of the handle can be inspected to determine the type of transaction to perform. The available options are:

- If the handle is arithmetically less than the value 256, the handle directly encodes one of the six predefined transaction types.
- If handle is arithmetically greater than the value 256, the handle is interpreted as a pointer to a bus controller routine that is used to implement the transaction.

These two forms of support are discussed in the following sections.

#### Specifying a Predefined Transaction Type

If a bus controller driver does not provide an implementation for **{vxbDevRegMap}( )**, a default transaction type for the eight **vxbRead*( )** and **vxbWrite*( )** routines is provided as shown in Table 2-1.

Table 2-1 **Available Transaction Types**

| Architecture | I/O Space | Memory Space |
|---|---|---|
| PowerPC | I/O access | Ordered memory access |
| All others (little-endian) | I/O access | Memory access |
| All others (big-endian) | Byte-swapped I/O access | Byte-swapped memory access |

These defaults suffice for the majority of target platforms. When these defaults are not appropriate, the bus controller must implement the **{vxbDevRegMap}( )** method in order to override the default access model.

If the bus controller provides the **{vxbDevRegMap}( )** method, this method is invoked by the **vxbRegMap( )** utility routine whenever a device driver residing on the controlled bus invokes **vxbRegMap( )**. **vxbRegMap( )** finds the bus controller instance, and calls the **func{vxbDevRegMap}( )** routine using the parameter list described in the routine prototype. The **index** parameter passed to the method is the index into the device **regBase[ ]**. Within **func{vxbDevRegMap}( )**, the driver must determine, based on the register window being mapped by the driver, which of the six predefined transaction types to select for access into that register window. Based on the desired access, **func{vxbDevRegMap}( )** creates a handle value that describes the requested access type as shown in Table 2-2.

Table 2-2 **Handle Values for Access Types**

| Type of Access | Value for Handle |
|---|---|
| Memory access | **VXB_HANDLE_MEM** |
| Ordered memory access | **VXB_HANDLE_ORDERED** |
| I/O access | **VXB_HANDLE_IO** |
| Byte-swapped memory access | **VXB_HANDLE_SWAP(VXB_HANDLE_MEM)** |
| Byte-swapped order memory access | **VXB_HANDLE_SWAP(VXB_HANDLE_ORDERED)** |
| Byte-swapped I/O access | **VXB_HANDLE_SWAP(VXB_HANDLE_IO)** |

The preprocessor macros that are used to create the handle values are found in *installDir*/**vxworks-6.***x*/**target/src/hwif/h/vxbus/vxbAccess.h**.

For example, if you want **func{vxbDevRegMap}( )** to perform strictly ordered memory accesses for all memory regions, and simple I/O operations for all I/O regions, you can implement the following code:

```
if (pChild->regBaseFlags[index] == VXB_REG_MEM)
    *pHandle = (void *) VXB_HANDLE_ORDERED;
else
    *pHandle = (void *) VXB_HANDLE_IO;
```

And if you need the same condition, but with the data values swapped for a big-endian processor, you can implement the following code:

```
if (pChild->regBaseFlags[index] == VXB_REG_MEM)
    *pHandle = (void *) VXB_HANDLE_SWAP(VXB_HANDLE_ORDERED);
else
    *pHandle = (void *) VXB_HANDLE_SWAP(VXB_HANDLE_IO);
```

**NOTE:** If your bus controller driver advertises a **func{vxbDevRegMap}( )** routine, it must return a correct handle value every time it is called. There is no way for this method to return a handle for only a subset of the device register windows.

### Providing a New Transaction Type

For some processor architectures, none of the six predefined transaction types work correctly. For example, a hardware architecture that uses keyhole memory cannot be supported by any of the six predefined transaction types. In this situation, your bus controller driver must implement its own access routine, and provide a pointer to that access routine to its subordinate driver instances. When these driver instances perform I/O operations to the target hardware using any of the **vxbRead*( )** or **vxbWrite*( )** routines, a callback is made to the routine provided by the bus controller driver, and this callback routine implements the request.

If your bus controller driver needs to provide a custom access routine, return a pointer to the driver access routine using the ***pHandle** parameter passed to **func{vxbDevRegMap}( )**. For example:

```
if (index == KEYHOLE_MEMORY_SPACE)
    *pHandle = (void *) driverAccessFunc;
else
    *pHandle = VXB_HANDLE_MEM;
```

The bus controller access routine has the following prototype:

```
LOCAL VOID driverAccessFunc
    (
    int     iodesc,     /* a descriptor for the requested IO operation */
    void *  pBuf,       /* pointer to buffer to read to or write from */
    UINT8 * offset      /* offset into the address space */
    )
```

The **driverAccessFunc( )** routine must be implemented to handle both read and write transactions, with bus widths of 8, 16, and 32 bits. If the bus supports 64-bit transactions, this routine must support 64-bit read and write transactions.

The **iodesc** parameter encodes the I/O operation to be performed. This parameter is broken apart into the **read/write** and **width** parameters using the following macros from *installDir*/**vxworks-6.***x*/**target/src/hwif/h/vxbus/vxbAccess.h**:

- **VXB_HANDLE_OP(iodesc)**—indicates whether it is a read operation or a write operation.
- **VXB_HANDLE_WIDTH(iodesc)**—indicates the size to read or write, in bytes.

**VXB_HANDLE_OP( )** can be used as follows:

```
If ( VXB_HANDLE_OP(iodesc) == VXB_HANDLE_OP_READ )
    {
    /* this is a read operation */
    ...
    }
else if ( VXB_HANDLE_OP(iodesc) == VXB_HANDLE_OP_WRITE )
    {
    /* this is a write operation */
    ...
    }
```

Once the transaction direction (read or write) and transaction width (1, 2, 4, or 8 bytes) is determined, **driverAccessFunc( )** performs whatever operations are required to complete the requested operation.

**NOTE:** Device drivers can perform the various **vxbRead*( )** or **vxbWrite*( )** operations while holding a spinlock. Because nesting of spinlocks is prohibited, **driverAccessFunc( )** cannot use spinlocks in its implementation.

### 2.3.7 {vxbIntDynaVecProgram}( )

Provides support for dynamic interrupt vector assignment.

```
STATUS func{vxbIntDynaVecProgram}
    (
    VXB_DEVICE_ID instID,
    struct vxbIntDynaVecInfo * dynaVec
    )
```

For more information on dynamic vectors, see *Programming Dynamic Vectors*, p.64.

## 2.4 Header Files

The following header files are typically used for all bus controller drivers:

```
#include <vxBusLib.h>
#include <hwif/vxbus/vxBus.h>
#include <hwif/vxbus/vxbPlbLib.h>
#include <hwif/vxbus/hwConf.h>
#include <hwif/util/hwMemLib.h>
#include "../h/vxbus/vxbAccess.h"
```

The following additional header files are used for PCI bus controller drivers:

```
#include <hwif/vxbus/vxbPciLib.h>
#include <drv/pci/pciConfigLib.h>
#include <drv/pci/pciAutoConfigLib.h>
#include <drv/pci/pciIntLib.h>
```

## 2.5 BSP Configuration

A general-purpose bus controller driver must obtain a substantial amount of information from the BSP before it can function correctly. This section describes the configuration fields that are expected by bus controller drivers.

The BSP configuration for a bus controller driver tends to be highly tailored to the needs of the particular bus controller in question. After several bus controller drivers have been developed for a particular bus type, common configuration

parameters become evident. At present, the majority of configuration parameters that are used for more than one bus controller driver are those that are used to configure the PCI bus. Table 2-3 lists the commonly used resources for the PCI bus along with a brief description of the resource. For more complete information about a particular resource, refer to the existing bus controller device drivers provided by Wind River.

Table 2-3 **Common Resources for a PCI Bus**

| Type | Name | Description |
|---|---|---|
| **HCF_RES_ADDR** | **mem32Addr** | Specifies the 32-bit pre-fetchable memory pool base address. |
| **HCF_RES_INT** | **mem32Size** | Specifies the 32-bit pre-fetchable memory pool size. |
| **HCF_RES_ADDR** | **memIo32Addr** | Specifies the 32-bit non-prefetchable memory pool base address. |
| **HCF_RES_INT** | **memIo32Size** | Specifies the 32-bit non-prefetchable memory pool size. |
| **HCF_RES_ADDR** | **io32Addr** | Specifies the 32-bit I/O pool base address. |
| **HCF_RES_INT** | **io32Size** | Specifies the 32-bit I/O pool size. |
| **HCF_RES_ADDR** | **io16Addr** | Specifies the 16-bit I/O pool base address. |
| **HCF_RES_INT** | **pciIo16Size** | Specifies the 16-bit I/O pool size. |

Individual bus controller drivers may define additional fields that are useful for the particular hardware. The resources that are defined by a particular bus controller driver are, by definition, tailored to the unique needs of the particular hardware.

### 2.5.1 PCI Configuration

In order to support the generation of PCI configuration cycles according to the PCI specification, a utility library is available to bus controller drivers. This library is located in *installDir*/**vxworks-6.***x*/**target/src/hwif/vxbus/vxPci.c**.

The choice of configuration method to use (method 0, method 1, or method 2) is often made configurable in the driver, so that a BSP can specify the configuration method (and perhaps also the correct addresses for the configuration registers used by these methods). If you want your bus controller driver to allow the BSP to choose the method to be used for the generation of PCI configuration cycles, be sure your driver exports the resource listed in Table 2-4.

Table 2-4 **Configuration Resources for PCI Bus**

| Type | Name | Description |
|---|---|---|
| HCF_RES_INT | **pciConfigMechanism** | A value between 0 and 2. |

### 2.5.2 PCI Autoconfiguration

PCI bus controllers often use the PCI autoconfiguration services that are included with VxWorks. If a bus controller driver uses this service, the related BSP must ensure that the resources required to support PCI autoconfiguration are defined in the BSP **hwconf.c** file. The following resources are used directly by **vxbPciAutoConfig( )**:

**autoIntRouteSet**
**bridgePostConfigFuncSet**
**bridgePreConfigFuncSet**
**cacheSize**
**fbbEnable**
**includeFuncSet**
**intAssignFuncSet**
**io16Addr**
**io16Size**
**io32Addr**
**io32Size**
**maxBusSet**
**maxLatAllSet**
**maxLatencyArgSet**
**maxLatencyFuncSet**
**mem32Addr**
**mem32Size**
**memIo32Addr**
**memIo32Size**
**msgLogSet**
**rollcallFuncSet**

In most cases, the bus controller does not need to manipulate these resources directly.

For details about the semantics of each of the PCI autoconfiguration resource, see the reference entry for **vxbPciAutoConfig( )**.

## 2.6 Available Utility Routines

The various utility services available to bus controller drivers are described in this section. The majority of the support described in this section is for PCI bus controller drivers, because this is the most prevalent bus type used in devices today.

### 2.6.1 PCI Configuration

A utility library is available to bus controller drivers to support the generation of PCI configuration cycles according to the PCI specification. The utility library is located in *installDir*/**vxworks-6.***x*/**target/src/hwif/vxbus/vxPci.c**. To initialize the library, the bus controller driver must call **vxbPciConfigLibInit( )**. The prototype for **vxbPciConfigLibInit( )** is as follows:

```
STATUS vxbPciConfigLibInit
    (
    struct vxbPciConfig *pPciConfig,
    int         pciConfAddr0,   /* used by method 1 & method 2 */
    int         pciConfAddr1,   /* used by method 1 & method 2 */
    int         pciConfAddr2,   /* used by method 2 only */
    int         pciConfigMech,  /* 1=method-1, 2=method-2 */
    int         pciMaxBus       /* Max number of sub-busses */
    )
```

**vxbPciConfigLibInit( )** initializes the memory pointed to by **pPciConfig**, so that this data structure can be used by the utility library when the utility library generates PCI configuration cycles. The bus controller driver is responsible for allocating the memory area used to store this data structure. After this data structure is initialized, the utility library can invoke the driver **{busCtlrCfgInfo}( )** method in order to retrieve the pointer to the data structure.

VxBus utility services that perform PCI autoconfiguration expect to be able to use PCI configuration utility services to perform the required configuration cycles. If PCI autoconfiguration is supported by the bus controller driver, the PCI configuration library must be properly initialized before autoconfiguration is performed.

For further details on the use of the PCI configuration library, see the reference entry for **vxbPci**.

### 2.6.2 PCI Autoconfiguration

On some hardware platforms, the devices that are available on the PCI bus are initialized before VxWorks begins operation. In other environments, device configuration is performed by VxWorks. Bus controller drivers for the PCI bus should support the configuration of the devices on the bus, unless they will only be executed in hardware environments where the PCI bus is configured before VxWorks starts.

The interface to PCI autoconfiguration is straightforward, consisting of only a single call to **vxbPciAutoConfig( )**. The prototype for **vxbPciAutoConfig( )** is:

```
STATUS vxbPciAutoConfig
    (
    VXB_DEVICE_ID busCtrlID
    )
```

The simplicity of this interface hides a great deal of configurability and complexity. As seen in *2.5.2 PCI Autoconfiguration*, p.19, PCI autoconfiguration requires a large number of configuration resources from the BSP. As such, many of the configuration requirements for PCI autoconfiguration are BSP requirements instead of bus controller driver requirements. If a bus controller driver makes use of PCI autoconfiguration, this creates an implicit dependency on the resources that PCI autoconfiguration requires.

### 2.6.3 vxbBusAnnounce( )

Each bus controller driver must inform VxBus that there is a bus downstream from it. This must occur early in the initialization process, typically during phase 1 initialization immediately after it allocates and initializes the per-driver data structures that it wants to maintain. The call to make VxBus aware of the downstream bus is **vxbBusAnnounce( )**. The prototype for **vxbBusAnnounce( )** is:

```
STATUS vxbBusAnnounce
    (
    VXB_DEVICE_ID     pBusDev,     /* bus controller */
    UINT32            busID        /* bus type */
    )
```

The **pBusDev** parameter refers to the bus controller instance, and is provided to the initialization routine that invokes **vxbBusAnnounce( )**. The second parameter (**busID**) identifies the type of bus being announced. Table 2-5 lists the available macros and their descriptions.

Table 2-5 **Bus Type Macros**

| Name | Description |
|---|---|
| **VXB_BUSID_PLB** | Processor Local Bus |
| **VXB_BUSID_PCI** | PCI |
| **VXB_BUSID_RAPIDIO** | RapidIO |
| **VXB_BUSID_MII** | Media Independent Interface (MII) |
| **VXB_BUSID_VIRTUAL** | virtual bus |

The list of supported bus types is likely to increase in future releases. For a complete list of the available bus types, refer to the **BUSID** definitions found in *installDir*/**vxworks-6.***x*/**target/h/hwif/vxbus/vxBus.h**.

### 2.6.4 vxbPciBusTypeInit( )

Once a PCI bus is configured so that subordinate devices on the bus are visible, each bus controller driver must call **vxbPciBusTypeInit( )**, to allow VxBus to perform any required connection operations to associate the discovered devices with their bus controller. The prototype for **vxbPciBusTypeInit( )** is:

```
STATUS vxbPciBusTypeInit
    (
    VXB_DEVICE_ID pBusDev
    )
```

The **pBusDev** parameter refers to the bus controller instance, and is provided to the initialization routine that invokes **vxbPciBusTypeInit( )**.

### 2.6.5 vxbPciDeviceAnnounce( )

Once **vxbPciBusTypeInit( )** is invoked, the bus controller driver invokes **vxbPciDeviceAnnounce( )** in order to make the discovered devices on the bus visible to VxBus. The prototype for **vxbPciDeviceAnnounce( )** is:

```
STATUS vxbPciDeviceAnnounce
    (
    VXB_DEVICE_ID pBusDev
    )
```

The **pBusDev** parameter refers to the bus controller instance, and is provided to the initialization routine that invokes **vxbPciDeviceAnnounce( )**.



## 2.7 Initialization

The initialization steps for a bus controller are similar, but not identical, to the initialization steps that occur for other drivers. This section discusses the various steps in the initialization of a bus controller driver.

Bus controller drivers must register themselves with VxBus during the boot process, as is the case with all VxBus drivers. The primary difference between bus controller drivers and other drivers is that bus controller drivers describe themselves differently in their **vxbDevRegInfo** initialization structure. Whereas most drivers declare themselves as being of type **VXB_DEVID_DEVICE**, bus controllers describe themselves as **VXB_DEVID_BUSCTRL**.

As with service drivers (see *A. Glossary*), bus controller drivers are initialized in three distinct phases. Typically, bus controller drivers are initialized during system boot. However, during early development, you may choose to delay the initialization of a bus controller driver until after the system is running. This provides a more robust debugging environment during bus controller driver development. When debugging is complete, be sure to restore your driver initialization to the earliest possible initialization phase. For more information, see *2.8 Debugging*, p.26.

The initialization of a bus controller can be thought of in terms of a driver's internal requirements, and of external requirements that are imposed on the driver by the VxBus bus controller driver model. Internal requirements are operations that the bus controller needs to perform in order to create a suitable run-time environment for itself. This typically includes:

- Allocating memory to hold per-instance data structures, and initializing them according to the driver's unique requirements.
- Reading in resource information from the environment, and programming the bus controller hardware to reflect the desired configuration.
- Initializing utility libraries that the bus controller driver will put to subsequent use within the driver.

Note that this list is not meant to be exhaustive. In some cases, device drivers will have unique requirements that occur along with the above examples.

In addition to the internal requirements of the bus controller driver, bus controller drivers have additional requirements in the way that they connect themselves to VxBus. This includes:

- Announcing the availability of the bus to VxBus.
- Scanning the bus (where possible), in order to find devices that are available on the bus, so that they can be paired with drivers to form additional instances.
- Performing bus-type specific operations, such as announcing the availability of a new PCI bus to VxBus.

VxWorks provides utility routines that help to provide support for a PCI bus. More specifically, utility services are provided to support PCI bus configuration. For more information on these utility services, see *2.6 Available Utility Routines*, p.20.

### 2.7.1 Initialization Example

While each bus controller driver may have unique initialization requirements, most requirements fall broadly into the steps outlined in this section.

For this example, the following steps are taken from the **g64120aPci.c** bus controller:

1. Allocate the per-instance data area used by the driver.

```
pDrvCtrl = hwMemAlloc (sizeof(*pDrvCtrl));
```

2. Query required resources from the BSP, and store them locally.

```
devResourceGet(pHcf, "maxBusSet",
               HCF_RES_INT, (void *)&pDrvCtrl->pciMaxBus);

devResourceGet(pHcf, "pciConfigMechanism",
               HCF_RES_INT, (void *)&pDrvCtrl->pciConfigMech)

/* etc. */
```

3. Initialize the support library used for PCI configuration handling.

```
vxbPciConfigLibInit(pDrvCtrl->pPciConfig, /*…*/ );
```

4. Initialize the bus controller hardware itself.

```
g64120aPciBridgeInit (pInst);
```

5. Inform VxBus about the availability of the new bus.

```
vxbBusAnnounce (pInst, VXB_BUSID_PCI);
```

6. If the BSP has requested PCI autoconfiguration, perform the autoconfigure now.

```
if (pDrvCtrl->autoConfig)
    vxbPciAutoConfig(pInst);
```

7. Complete VxBus initialization.

```
vxbPciBusTypeInit (pInst);
vxbSubDevAction (pInst,vxbUpdateDeviceInfo,0,
                 VXB_ITERATE_ORPHANS);
vxbPciDeviceAnnounce(pInst);
```

The first-pass driver initialization routine is intended primarily for bus controller devices. Bus controller devices can 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 **vxbBusAnnounce( )**. The bus controller device driver is responsible for device enumeration[1]. Depending on the system configuration options, one of three versions can 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 **vxbDeviceAnnounce( )**.

The following sections describe the routines that are provided by VxBus for registration of devices and bus types.

### vxbBusAnnounce( )

**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.

```
STATUS vxbBusAnnounce
    (
    VXB_DEVICE_ID pBusDev, /* bus controller */
    UINT32        busID    /* bus type */
    )
```

1. PCI Bus controller drivers call **vxbPciDeviceAnnounce( )**, which provides device enumeration on behalf of the caller. Therefore, no additional code is required in the PCI bus controller driver to perform device enumeration.

#### vxbDeviceAnnounce( )

**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. The prototype is:

```
STATUS vxbDeviceAnnounce
    (
    VXB_DEVICE_ID pBusDev
    )
```

#### vxbDevStructAlloc( )

Each bus controller driver is responsible for enumerating the devices on the bus and announcing them to VxBus. The **vxbDevStructAlloc( )** routine allocates a device structure. The bus controller driver fills in the fields of the structure and then announces the newly discovered device to VxBus with a call to **vxbDeviceAnnounce( )**. The prototype is:

```
VXB_DEVICE_ID vxbDevStructAlloc(void);
```

#### vxbDevStructFree( )

**vxbDevStructFree( )** returns the device structure to the pool, making it available for future device allocation. The prototype is:

```
void vxbDevStructFree(VXB_DEVICE_ID devID);
```

## 2.8 Debugging

Bus controller drivers can be quite complex, and by design they should initialize themselves as early as possible during the VxWorks boot process, so that devices on the bus can themselves be initialized and become available to the operating system. However, because no devices are available that can be used to aid in the debug process, the run-time environment that exists when a bus controller driver is doing its initialization is very limited. For example, because no console or other

serial devices are available, services like **logMsg( )** are not helpful in the debug process.

Fortunately, it is not mandatory that a bus controller driver be initialized during the first phase of VxBus initialization. During the development process, you may wish to delay the initialization of a bus controller driver until much later in the system boot process, so that operating system services (such as **printf( )** and **logMsg( )**) can be used to aid in the debugging process.

For details about deferring initialization and enabling debug support within a driver, refer to *VxWorks Device Driver Developer's Guide (Vol.1): Development Strategies*.

# 3 Direct Memory Access Drivers


3.1 Introduction 29

3.2 Overview 30

3.3 VxBus Driver Methods 30

3.4 Header Files 32

3.5 BSP Configuration 32

3.6 Available Utility Routines 33

3.7 Initialization 33

3.8 DMA System Structures and Routines 33

3.9 Debugging 36


## 3.1 Introduction

This chapter describes direct memory access (DMA) drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 3.2 Overview

Some hardware designs include a general-purpose direct memory access (DMA) engine that handles DMA accesses from, or to, external devices or from memory to memory. These DMA engines are often found integrated in system-on-chip CPU designs. The DMA driver class provides a standard method for presenting the services of these DMA engines to other drivers in the system.

The **vxbDmaLib** library is provided for drivers that wish to use a DMA engine. The routines provided by this DMA library are **vxbDmaChanAlloc( )** and **vxbDmaChanFree( )**. (For more information on these routines, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.)

## 3.3 VxBus Driver Methods

The routines provided by **vxbDmaLib** make use of three VxBus driver methods:

- **{vxbDmaResourceGet}( )**
- **{vxbDmaResourceRelease}( )**
- **{vxbDmaResDedicatedGet}( )**

DMA drivers provide access to their services by associating routines with these methods.

### 3.3.1 {vxbDmaResourceGet}( )

The **{vxbDmaResourceGet}( )** method is used by the DMA library to allocate a DMA channel on the device managed by the DMA driver. The prototype is as follows:

```
STATUS {vxbDmaResourceGet}
    (
    VXB_DEVICE_ID      pInst,
    VXB_DEVICE_ID      pReqDev,
    VXB_DMA_REQUEST *  pReq
    )
```

In this prototype, **pInst** refers to the DMA device itself, **pReqDev** refers to the device requesting a DMA channel, and **pReq** is a pointer to a structure describing the desired attributes for the DMA channel.

The **VXB_DMA_REQUEST** structure is defined in *installDir*/**vxworks-6.***x*/**target/src/hwif/h/util/vxbDmaDriverLib.h** as follows:

```
typedef struct vxbDmaRequest
    {
    VXB_DEVICE_ID       instance;           /* DMA requestor device id */
    UINT32              minQueueDepth;      /* minimum queue depth
                                               requested */
    UINT32              flags;              /* flags used during DMA
                                               allocation */
    VXB_DMA_RESOURCE_ID pChan;              /* DMA channel id */
    void *              pDedicatedChanInfo; /* dedicated channel
                                               information */
   } VXB_DMA_REQUEST;
```

This structure largely corresponds to the parameters passed to **vxbDmaChanAlloc( )**. DMA device drivers normally select a DMA channel based on **minQueueDepth** and **flags**, and return a pointer to the channel in **pChan**. Device drivers making a call to the DMA driver's channel allocation code, whether through **func{vxbDmaResourceGet}( )** or through **func{vxbDmaResDedicatedGet}( )** can optionally pass a pointer to a structure containing information specific to the expected DMA channel dedicated to the requestor. The DMA driver can make use of this information to set up a dedicated DMA channel.

### 3.3.2 {vxbDmaResourceRelease}( )

The **{vxbDmaResourceRelease}( )** method is used by the DMA library to free a DMA channel on the device managed by the DMA driver. The prototype is as follows:

```
STATUS {vxbDmaResourceRelease}
    (
    VXB_DEVICE_ID       pInst,
    VXB_DMA_RESOURCE_ID pChan
    )
```

In most cases, the only requirement for the driver is to free the particular DMA channel allocated to the device identified by **pChan**. **pInst** refers to the VxBus device ID of the DMA device.

### 3.3.3 {vxbDmaResDedicatedGet}( )

The **{vxbDmaResDedicatedGet}( )** method is used by the DMA library to allocate a DMA channel dedicated to the particular device that called the method. This method is functionally similar to **{vxbDmaResourceGet}( )**. However, due to

hardware constraints or other considerations, you may wish to use it to ensure that particular devices are allocated to particular channels. This can be accomplished, for example, by checking the device name associated with the device instance identified by **pReqDev**, or by checking information passed in using the **pDedicatedChanInfo** member of **pReq**. The prototype is as follows:

```
STATUS {vxbDmaResDedicatedGet}
    (
    VXB_DEVICE_ID      pReqDev,
    VXB_DMA_REQUEST *  pReq
    )
```

## 3.4 Header Files

DMA drivers must include the following header files:

```
#include <hwif/util/vxbDmaLib.h>
#include "../h/util/vxbDmaDriverLib.h"
```

Other drivers that wish to use **vxbDmaLib** may need to include the following:

```
#include <hwif/util/vxbDmaLib.h>
```

These drivers may also need to include the header files for specific DMA drivers, in order to use the dedicated channel functionality.

## 3.5 BSP Configuration

DMA drivers do not typically require configuration information from a BSP that is above and beyond the normal device-specific information provided for all drivers. For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.

## 3.6 Available Utility Routines

There are no class-specific utility routines required or available for DMA drivers.



## 3.7 Initialization

The initialization of DMA device drivers is generally device-specific. Initialization should be completed before or during VxBus initialization phase 2, so that other drivers are guaranteed that **vxbDmaLib** is available during initialization phase 3.

## 3.8 DMA System Structures and Routines

The routines and methods described in previous sections make use of **VXB_DMA_RESOURCE_ID** to identify a particular DMA channel. This identifier is a pointer to a **vxbDmaResource** structure, and is defined as follows:

```
struct vxbDmaResource
    {
    struct vxbDmaFuncs  dmaFuncs;/* structure holding dma
                                    function pointers */
    void *              pDmaChan;/* channel specific data-used by DMA
                                    driver */
    VXB_DEVICE_ID       dmaInst; /* dma engine instance ID */
    };
```

The **dmaFuncs** member of this structure contains function pointers that are used for various DMA operations. Device drivers can access these routines through the **VXB_DMA_RESOURCE_ID** identification returned to them using a call to **vxbDmaChanAlloc( )**. These function pointers should be filled in by DMA drivers. Depending on the flags argument passed to the **vxbDmaChanAlloc( )** routine, **vxbDmaLib** may initialize the read and write routines with software versions. The **vxbDmaFuncs** structure is defined in **vxbDmaLib.h**, and contains pointers to the routines described in the following sections.

### 3.8.1 (*dmaRead)( )

**(*dmaRead)( )** queues a read from the buffer or register on the device to a buffer in system memory. Control is returned immediately to the caller, with an **OK** status if the transaction can be queued, or **ERROR** if the DMA device queue is full. **pDmaComplete** and **pArg** can be used to specify a callback routine for when the transaction is complete.

```
STATUS (*dmaRead)
    (
    VXB_DMA_RESOURCE_ID     dmaChan,
    char *                  src,
    char *                  dest,
    int                     transferSize,
    int                     unitSize,
    UINT32                  flags,
    pVXB_DMA_COMPLETE_FN    pDmaComplete,
    void *                  pArg
    );
```

### 3.8.2 (*dmaReadAndWait)( )

**(*dmaReadAndWait)( )** is similar to **(*dmaRead)( )** except that control is not returned to the caller until the transaction is complete.

```
STATUS (*dmaReadAndWait)
    (
    VXB_DMA_RESOURCE_ID     dmaChan,
    char *                  src,
    char *                  dest,
    int *                   pTransferSize,
    int                     unitSize,
    UINT32                  flags
    );
```

### 3.8.3 (*dmaWrite)( )

**(*dmaWrite)( )** queues a write from the buffer or register on the device, to a buffer in system memory. Control is returned immediately to the caller, with an **OK** status if the transaction can be queued, or **ERROR** if the DMA device queue is full. **pDmaComplete** and **pArg** can be used to specify a callback routine for when the transaction is complete.

```
STATUS (*dmaWrite)
    (
    VXB_DMA_RESOURCE_ID     dmaChan,
    char *                  src,
    char *                  dest,
    int                     transferSize,
    int                     unitSize,
    UINT32                  flags,
    pVXB_DMA_COMPLETE_FN    pDmaComplete,
    void *                  pArg
    );
```

### 3.8.4 (*dmaWriteAndWait)( )

**(*dmaWriteAndWait)( )** is similar to **(*dmaWrite)( )** except that control is not returned to the caller until the transaction is complete.

```
STATUS (*dmaWriteAndWait)
    (
    VXB_DMA_RESOURCE_ID     dmaChan,
    char *                  src,
    char *                  dest,
    int *                   pTransferSize,
    int                     unitSize,
    UINT32                  flags
    );
```

### 3.8.5 (*dmaCancel)( )

**(*dmaCancel)( )** cancels a read or write operation that was previously started on a given channel. This prevents any further I/O from occurring on the channel until a new read or write operation is queued.

```
STATUS (*dmaCancel)
    (
    VXB_DMA_RESOURCE_ID     dmaChan
    );
```

### 3.8.6 (*dmaPause)( )

**(*dmaPause)( )** pauses a DMA channel that previously started a transfer. Pausing a channel allows the caller to safely manipulate any underlying DMA descriptor or buffer structures associated with the channel without cancelling the DMA operation completely. A paused channel can be resumed with **(*dmaResume)( )**.

```
STATUS (*dmaPause)
    (
    VXB_DMA_RESOURCE_ID     dmaChan
    );
```

### 3.8.7 (*dmaResume)( )

**(*dmaResume)( )** resumes a DMA channel that has been paused, or which has gone idle.

```
STATUS (*dmaResume)
    (
    VXB_DMA_RESOURCE_ID     dmaChan
    );
```

### 3.8.8 (*dmaStatus)( )

**(*dmaStatus)( )** returns the status of the specified DMA channel. The valid return value are: **DMA_NOT_USED**, **DMA_IDLE**, **DMA_RUNNING**, or **DMA_PAUSED**.

```
int (*dmaStatus)
    (
    VXB_DMA_RESOURCE_ID     dmaChan
    );
```

## 3.9 Debugging

Because they can be tested when the VxWorks system is fully initialized, debugging DMA drivers is generally straightforward. When debugging DMA drivers, the full debug capabilities of VxWorks, as well as conventional instrumentation techniques such as **logMsg( )**, can be used effectively.

The only complicating factor is that DMA drivers cannot be tested in a vacuum. Because they provide a service to other drivers in the system, they must be tested with another driver. For debugging purposes, you may wish to write a dummy driver that calls the routines in **vxbDmaLib** to allocate a DMA channel and initiate mock DMA transfers.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 4 Interrupt Controller Drivers


4.1 Introduction 38
4.2 Overview 38
4.3 VxBus Driver Methods 41
4.4 Header Files 45
4.5 BSP Configuration 45
4.6 Available Utility Routines 51
4.7 Initialization 56
4.8 Interrupt Controller Typologies and Hierarchies 57
4.9 Interrupt Priority 58
4.10 ISR Dispatch 59
4.11 Managing Dynamic Interrupt Vectors 62
4.12 Internal Representation of Interrupt Inputs 65
4.13 Multiprocessor Issues with VxWorks SMP 66
4.14 Debugging 73

## 4.1 Introduction

This chapter describes interrupt controller drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 4.2 Overview

This chapter provides information on interrupt identification, driver responsibilities, interrupt controller configurations, dynamic vector assignment, and multiprocessing systems as they relate to VxBus model interrupt controller drivers. This section describes these topics briefly. The remainder of the chapter provides the detailed information necessary to understand VxBus model interrupt controller drivers.

Within the VxBus framework, interrupt controller hardware management can be implemented with a VxBus driver.

**NOTE:** For use with the VxWorks SMP product, the interrupt controller code must be implemented as a VxBus driver.

Interrupt controller drivers are among the most difficult device drivers to create, debug, and maintain. When writing a VxBus interrupt controller driver, Wind River recommends that you first understand the information in the *VxWorks Device Driver Developer's Guide, Volume 1*. Then, read and understand this chapter. Finally, review the VxBus interrupt controller drivers provided by Wind River to find the one that most closely matches the hardware you are working with, use that driver as a model for your development.

**NOTE:** The OpenPIC interrupt controller driver, **vxbEpicIntCtlr.c**, and the PowerPC CPU-specific interrupt controller driver, **vxbPpcIntCtlr.c** provided by Wind River are generally appropriate to use as models for interrupt controller driver development. However, because these drivers are subject to Wind River guidelines for backward compatibility, they may include code that is not necessary for your development situation. In this case, you may wish to create an entirely new interrupt controller driver in order to simplify the driver code.

### Interrupt Identification

Within the context of VxBus, an interrupt is considered to be an entity specific to the device that generates the interrupt. That is, in VxBus, all interrupts are identified by the VxBus device and an interrupt index. This uniquely identifies every interrupt source on the system based on what generates the interrupt.

From the perspective of an interrupt controller, you must also refer to interrupts by the input pins on which the interrupt arrives. When discussing interrupt controllers, this is referred to as the *interrupt input*.

Interrupt identification is discussed further in *4.12 Internal Representation of Interrupt Inputs*, p.65.

### Interrupt Controller Driver Responsibilities

The interrupt controller driver is responsible for maintaining interrupt routing information, managing interrupt input characteristics such as trigger type (edge trigger or level trigger), trigger value (active high or active low), and other characteristics of the interrupt source.

Interrupt controller drivers are also responsible for maintaining ISRs for each interrupt input, and for the argument that is passed to each ISR. The library **vxbIntCtlrLib** provides routines to help manage ISR connections. The library attempts to dispatch ISRs in the most efficient manner possible. When a single ISR is connected, the ISR is dispatched directly. When multiple ISRs are connected, **vxbIntCtlrLib** creates a chain of ISR handlers to call when an interrupt occurs.

When any driver makes a call to **vxbIntConnect( )**, each interrupt controller on the system is given a chance to claim the interrupt. Once the interrupt is claimed, that interrupt controller is responsible for managing the interrupt as required by the hardware and as directed by calls to **vxbIntEnable( )**, **vxbIntDisable( )**, and **vxbIntDisconnect( )**. These calls map into interrupt controller methods.

Driver responsibilities are discussed further in *4.3 VxBus Driver Methods*, p.41.

### Interrupt Controller Configurations

Many CPUs have the ability to wire multiple interrupts directly to the CPU. Other CPUs can wire only a single interrupt directly to the CPU, and any interrupt management must be handled by an external interrupt controller device. Other CPUs have the capability for interrupts to be indicated as messages on a separate

bus of some kind, so that no interrupts need to be hard-wired directly to the CPU. Some external interrupt controllers also provide this functionality separate from the CPU. VxBus interrupt controller drivers support all of these configurations.

Interrupt controllers can also have a hierarchy of connectivity, where the inputs of some interrupt controllers are connected to the outputs of other interrupt controllers.

Interrupt controller configurations are discussed further in *4.8 Interrupt Controller Typologies and Hierarchies*, p.57.

#### Dynamic Vectors

Some hardware allows dynamic assignment of interrupt identifiers. Individual bus types, such as PCI, may define a bus-specific mechanism for handling dynamic vectors. For a PCI bus, this includes MSI and MSI-X. Even without any bus-specific dynamic vector assignment, individual devices can provide a mechanism for software to write a vector into a device register, to be used when the device generates an interrupt. In modern hardware, this sometimes happens when an interrupt controller is part of the same multifunction chip as other devices. One example of this is the OpenPIC timer. In this case, the timer device sits on the same chip as the interrupt controller and the hardware requires you to write a register on the timer device that contains the interrupt input number on the interrupt controller device.

Some VxBus interrupt controller drivers handle dynamic vector assignment for both of these conditions.

Dynamic vector management is discussed further in *4.11 Managing Dynamic Interrupt Vectors*, p.62.

#### Interrupt Controller Drivers and Multiprocessing

There are several areas of functionality relevant to multiprocessor systems that are handled by the interrupt controller driver. This includes assignment of a given interrupt to a specified CPU, and generation and management of interprocessor interrupts (IPIs).

Multiprocessor issues are discussed further in *4.13 Multiprocessor Issues with VxWorks SMP*, p.66.

## 4.3 VxBus Driver Methods

There are three groups of driver methods relevant to interrupt controller drivers. The first group is required for basic interrupt controller functionality. The second group deals with issues related to dynamic vector assignment. The last group deals with issues related to multiprocessor systems.



### 4.3.1 Basic Methods

The methods listed in this section are required for basic interrupt controller functionality.

#### {vxbIntCtlrConnect}( )

The **func{vxbIntCtlrConnect}( )** routine configures the hardware for the specified interrupt and attaches the supplied routine and argument to the appropriate interrupt input.

```
LOCAL STATUS func{vxbIntCtlrConnect}
    (
    VXB_DEVICE_ID   pIntCtlr,             /* interrupt controller VxBus
                                             device ptr */
    VXB_DEVICE_ID   pInst,                /* interrupt source VxBus
                                             device ptr */
    int             indx,                 /* device interrupt index */
    void            (*pIsr)(void * pArg), /* routine to be called */
    void *          pArg,                 /* parameter to be passed
                                             to routine */
    int *           pInputPin             /* found input pin for specified
                                             device */
    )
```

#### {vxbIntCtlrDisconnect}( )

The **func{vxbIntCtlrDisconnect}( )** routine disconnects the specified ISR and argument from the interrupt input and disables the interrupt input if it is not shared with other ISRs.

```
LOCAL STATUS func{vxbIntCtlrDisconnect}
    (
    VXB_DEVICE_ID   pIntCtlr,             /* interrupt controller VxBus
                                             device ptr */
```

```
    VXB_DEVICE_ID   pInst,                /* interrupt source VxBus
                                             device ptr */
    int             indx,                 /* device interrupt index */
    void            (*pIsr)(void * pArg), /* routine to be called */
    void *          pArg                  /* parameter to be passed
                                             to routine */
    )
```

### {vxbIntCtlrEnable}( )

The **func{vxbIntCtlrEnable}( )** enables the interrupt input and marks the specified ISR as enabled.

```
LOCAL STATUS func{vxbIntCtlrEnable}
    (
    VXB_DEVICE_ID   pIntCtlr,             /* interrupt controller VxBus
                                             device ptr */
    VXB_DEVICE_ID   pInst,                /* interrupt source VxBus
                                             device ptr */
    int             indx,                 /* device interrupt index */
    void            (*pIsr)(void * pArg), /* routine to be called */
    void *          pArg                  /* parameter to be passed
                                             to routine */
    )
```

### {vxbIntCtlrDisable}( )

The **func{vxbIntCtlrDisable}( )** marks the specified ISR as disabled. If there are no other enabled ISRs chained to the same interrupt input, the routine disables the input.

```
LOCAL STATUS func{vxbIntCtlrDisable}
    (
    VXB_DEVICE_ID   pIntCtlr,             /* interrupt controller VxBus
                                             device ptr */
    VXB_DEVICE_ID   pInst,                /* interrupt source VxBus
                                             device ptr */
    int             indx,                 /* device interrupt index */
    void            (*pIsr)(void * pArg), /* routine to be called */
    void *          pArg                  /* parameter to be passed
                                             to routine */
    )
```

### 4.3.2 Dynamic Vector Methods

The method listed in this section is used for dynamic vector assignment.

#### {vxbIntDynaVecConnect}( )



The **{vxbIntDynaVecConnect}( )** method allows a driver to request that multiple interrupts be assigned for use by the caller's device and a specified ISR/argument be attached to each.

When called, the **func{vxbIntDynaVecConnect}( )** routine causes interrupt vectors to be assigned to the requested device and connects the specified ISRs and arguments to the interrupts.

```
LOCAL STATUS func{vxbIntDynaVecConnect}
    (
    VXB_DEVICE_ID       pIntCtlr,
    VXB_DEVICE_ID       pInst,
    int vecCount,
    struct vxbIntDynaVecInfo* dynaVec
    )
```

Dynamic vector assignment currently requires that the driver call a special routine to assign dynamic vectors, or that the BSP be configured to use dynamic vectors. For more information, see *4.5 BSP Configuration*, p.45.

### 4.3.3 Multiprocessor Methods

The methods listed in this section are available for use in multiprocessor systems.

#### {vxbIntCtlrIntReroute}( )

The **func{vxbIntCtlrIntReroute}( )** routine reroutes a specified interrupt from the CPU to which it is currently routed, to the CPU specified by the **destCpu** argument.

```
LOCAL STATUS func{vxbIntCtlrIntReroute}
    (
    VXB_DEVICE_ID       pInst,
    int                 index,
    cpuset_t            destCpu
    )
```

The interrupt is specified by the device and index indicated in the arguments. All interrupts connected to the same input are rerouted together.

#### {vxbIntCtlrCpuReroute}( )

The **func{vxbIntCtlrCpuReroute}( )** routine reroutes interrupts from the CPU to which they are currently routed, to the CPU or CPUs specified by the **destCpu** argument.

```
LOCAL STATUS func{vxbIntCtlrCpuReroute}
    (
    VXB_DEVICE_ID       pInst,
    void *              destCpu
    )
```

While **{vxbIntCtlrIntReroute}( )** is specific to a single interrupt input, **{vxbIntCtlrCpuReroute}( )** routes all interrupts configured for a different CPU to that CPU as a block. That is, if the BSP configures four interrupt inputs as routed to CPU 1 using the CPU routing table in **hwconf.c**, then a single call to **func{vxbIntCtlrCpuReroute}( )** must reroute all four of those interrupts that are routed to CPU 1.

#### {vxIpiControlGet}( )

Interprocessor interrupts (IPIs) are used for various purposes in multiprocessor systems. The **func{vxIpiControlGet}( )** routine returns a pointer to a structure, **VXIPI_CTRL_INIT**, containing information to manage IPIs.

```
LOCAL VXIPI_CTRL_INIT * func{vxIpiControlGet}
    (
    VXB_DEVICE_ID pInst,
    void * pArg
    )
```

For more information on IPIs, see *4.13.2 Interprocessor Interrupts*, p.68.

## 4.4 Header Files

There are two header files available to VxBus interrupt controller drivers.



#### vxbIntrCtlr.h

The file **vxbIntrCtlr.h** contains information needed for retrieving interrupt routing information from the BSP. Include this file as follows:

```
#include <hwif/vxbus/vxbIntrCtlr.h>
```

#### vxbIntCtlrLib.h

Interrupt controller drivers should also include **vxbIntCtlrLib.h** when they use **vxbIntCtlrLib** routines, which is strongly recommended. This header file is located in *installDir*/**vxworks-6.***x*/**target/src/hwif/intCtlr**, therefore Wind River interrupt controller drivers simply include it using quotation marks.

```
#include "vxbIntCtlrLib.h"
```

When a third-party interrupt controller driver is released, the driver should be located in the directory *installDir*/**vxworks-6.***x*/**target/3rdparty/***vendor/driver*. In order to include **vxbIntCtlrLib.h**, the makefile in this directory should be modified to add **-I$(TGT_DIR)/src/hwif/intCtlr** to the **EXTRA_INCLUDE** macro as follows:

```
EXTRA_INCLUDE=-I$(TGT_DIR)/h -I$(TGT_DIR)/src/hwif/intCtlr
```

This modification allows third-party interrupt controller drivers to use angle brackets in the include line:

```
#include <vxbIntCtlrLib.h>
```

## 4.5 BSP Configuration

The device registers for almost all interrupt controllers are located logically on the processor bus. For this reason, interrupt controller drivers almost always need to have entries in the BSP **hwconf.c** file. Interrupt controller drivers require the standard **hwconf.c** entries. (For more information about **hwconf.c**, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.) However,

interrupt controller drivers also require additional entries to describe interrupt routing and configuration. The remainder of this section discusses these additional requirements.

Interrupt descriptions are represented by a series of tables in **hwconf.c**. For each table required by a given interrupt controller driver, a resource entry containing a pointer to the head of the table and a resource entry containing the size of the table are included in the interrupt controller's resource table.

The tables in **hwconf.c** include:

- an interrupt routing table, **input**, which lists devices that are connected to a specific interrupt input on the interrupt controller
- a priority table, **priority**, which lists the non-default priority of individual interrupt inputs
- a dynamic vector table, **dynamicInterrupt** or **dynamicInterruptTable**, which lists devices requiring dynamic vector assignment
- a CPU routing table, **cpuRoute**, which lists devices routed to processors other than the boot processor in a multiprocessor system
- a cross connect routing table, **crossBar**, that lists the input pin to output pin routing for each interrupt source to the interrupt controller

You may wish to use additional tables. This option is available, but not recommended by Wind River.

### 4.5.1 Interrupt Input Table

Interrupt input information is obtained from tables in the BSP **hwconf.c** file. The input information is represented by a table of structures of type **intrCtlrInputs**, which is defined in *installDir*/**vxworks.6.***x*/**target/h/hwif/vxbus/vxbIntrCtlr.h**. While you may not need to know the representation of information in **hwconf.c** to develop your driver, you do need to know this information in order to test the driver.

```
/*
 * intrCtlrInputs structure is used to associate a device with
 * the interrupt controller to which the device's interrupt
 * output is connected.  Note that multiple devices can be
 * connected to a single input pin; therefore, multiple
 * intrCtlrInputs table entries can be present for a single
 * input pin.  Also note that some input pins may not be
 * connected, which may leave holes in the table, where no
 * entry is present for a specific input pin.
 */
```

```
struct intrCtlrInputs
    {
    int         inputPin;
    char *      drvName;
    int         drvUnit;
    int         drvIndex;
    };
```



A pointer to the beginning of the table is provided in the device resource list with the name **input** of type **HCF_RES_ADDR**. The size of the table is provided with a resource name **inputTableSize** of type **HCF_RES_INT**.

When your driver initializes the ISR handle, **vxbIntCtlrLib** reads this table automatically.

Each device interrupt output that is connected to an interrupt input is listed in the table. In the following example, modified from the **hpcNet8641** BSP, macros are expanded to show the numeric values. Other modifications have been made for demonstration purposes.

```
struct intrCtlrInputs epicInputs[] = {
    { 19, "pciSlot", 0, 0 },
    { 20, "pciSlot", 0, 1 },
    { 21, "pciSlot", 0, 2 },
    { 22, "pciSlot", 0, 3 },
    { 22, "pciexpress", 0, 0 },
    { 38, "ns16550", 0, 0},
    { 24, "ns16550", 1, 0 },
    { 25, "mottsec", 0, 0 },
    { 26, "mottsec", 0, 1 },
    { 30, "mottsec", 0, 2 },
    { 31, "mottsec", 1, 0 },
    { 32, "mottsec", 1, 1 },
    { 36, "mottsec", 1, 2 },
    { 27, "mottsec", 2, 0 },
    { 28, "mottsec", 2, 1 },
    { 29, "mottsec", 2, 2 },
    { 33, "mottsec", 3, 0 },
    { 34, "mottsec", 3, 1 },
    { 35, "mottsec", 3, 2 },
    { 68, "ipi", 0, 0 }
    { 68, "dshmBusCtlr8641", 0, 0 }
};
```

Multiple interrupt sources can be listed for a single interrupt input. In this example, note that the PCI slot 0 interrupt output 3 (int-D) is wired to the same interrupt controller input pin, 22, as the PCI Express interrupt output. This is indicated by the following lines:

```
    { 22, "pciSlot", 0, 3 },
    { 22, "pciexpress", 0, 0 },
```

The order that input pins are listed in is not relevant. In this example, the order of inputs has been rearranged so that the outputs of each interrupt source are grouped together. This means that the input pin numbering shown in the example is sorted by input pin. In the released version, the entries are not sorted.

Note that the same interrupt input can be used for more than one purpose. In the example, **"ipi"** and **"dshmBusCtlr8641"** are both connected to the same interrupt input. These two interrupts do not occur in the same configuration. However, even when they do occur in the same configuration, they can both be present in the same image, with no functional adverse effects.

### 4.5.2 Dynamic Vector Table

There are several kinds of dynamic vectors that can exist in a system (see *4.11 Managing Dynamic Interrupt Vectors*, p.62) including bus-specific dynamic vectors such as message signalled interrupts (MSIs) on PCI bus types, as well as custom dynamic vector support on some multifunction chips that contain an interrupt controller device. The interrupt controller driver for systems that support dynamic vector table functionality must be created to support dynamic vectors.

In general, there are two ways of configuring a system to perform dynamic vector assignment. The first way is for your device driver to call a special routine to install dynamic vectors. If you need to install multiple ISRs to dynamic vectors, you must use this interface. This option is handled by a special driver method to support dynamic vector assignment.

The second way is available from the BSP. In this case, the driver does need to understand the implementation. The interrupt input is configured in the input table in the BSP **hwconf.c** file where it is specified using a device name, a device unit number, and a device interrupt output. The indication that this is a dynamically assigned vector is shown by the use of **VXB_INTR_DYNAMIC** as the input pin.

For example, in order to specify that the PCI network device **yn0** output 0 should use a dynamically generated vector, the following line is included in the table specified with the **input** resource.

```
{ VXB_INTR_DYNAMIC, "yn", 0, 0 },
```

Any number of interrupt sources can be specified with **VXB_INTR_DYNAMIC** as the input pin, and each of them must have a dynamic vector assigned when the ISR is connected.

### 4.5.3 CPU Routing Table

In some cases, the interrupt controller driver is expected to be used in a multiprocessor environment, and the interrupt controller hardware is able to route interrupt inputs to processors other than the boot processor. In this environment, the BSP can be configured to route interrupt inputs to the additional processors. The following discussion focuses on the optional VxWorks SMP product, but may be applicable to asymmetric multiprocessing (AMP) environments as well.



Routing interrupt inputs to non-default CPUs is configured by the presence of a table in the interrupt controller resources list. Because the interrupt controller can only route inputs, and because all interrupt sources on the same input must be routed to the same CPU at the same time, the interrupt inputs are identified by interrupt input pin number and not the normal interrupt identification mechanism consisting of **VXB_DEVICE_ID** and the interrupt output. The structure used for this is the **intCtlrCpu** structure, which is defined in *installDir*/**vxworks-6.***x*/**target/h/hwif/vxbus/vxbIntrCtlr.h** as follows:

```
/*
 * intrCtlrCpu is used on SMP systems only.  It indicates
 * which CPU the interrupt controller should route the
 * input pin to
 */

struct intrCtlrCpu
    {
    int         inputPin;
    int         cpuNum;
    };
```

The following is an example of the CPU interrupt routing table taken from the **hpcNet8641** BSP, with macros left in place for clarity. It has been created as an example with minimal effects on system configuration and performance, and not for maximizing interrupt performance.

```
struct intrCtlrCpu epicCpu[] = {
    { EPIC_TSEC3ERR_INT_VEC, 1 },
    { EPIC_TSEC1ERR_INT_VEC, 1 },
    { EPIC_TSEC4ERR_INT_VEC, 1 },
    { EPIC_TSEC2ERR_INT_VEC, 1 }
};
```

### 4.5.4 Interrupt Priority

Within the VxBus interrupt controller design, each interrupt input can be assigned a priority. This section describes the tables used to assign interrupt priority to specific interrupt inputs at the interrupt controller. For more information on interrupt priority and how it affects interrupt controller drivers, see *4.9 Interrupt Priority*, p.58.

As with other interrupt input configurations, the priority of interrupt inputs is defined as a table in the interrupt controller resource table, with a resource entry named **priority** to point to the first element of the table, and an entry named **priorityTableSize** to show the size of the priority table. The table is of type **intrCtlrPriority**, which is defined in *installDir*/**vxworks-6.***x*/**target/h/hwif/vxbus/vxbIntrCtlr.h** as follows:

```
/*
 * intrCtlrPriority is used to set the priority of
 * a specified input pin on an interrupt controller
 */

struct intrCtlrPriority
    {
    int         inputPin;
    int         priority;
    };
```

The default value of 15 does not need to be specified, but all other values are required. The following is an example of the priority assignment for the EPIC interrupt controller, as used in the **hpcNet8641** BSP.

```
struct intrCtlrPriority epicPriority[] = {
    { EPIC_DUART2_INT_VEC, 100 },
    { EPIC_DUART_INT_VEC,  100 }
};
```

### 4.5.5 Crossbar Routing Table

For crossbar interrupt controllers, there is an additional structure definition and table to hold the input pin and correlation to the output pin. If not specified, every input pin is assigned to the default output, which is output zero unless otherwise documented in the interrupt controller driver documentation. Do not route a single input pin to multiple output pins. This results in unpredictable behavior.

```
struct intrCtlrCpu
    {
    int     inputPin;
    int     outputPin;
    };
```

## 4.6 Available Utility Routines

There are a number of utility routines available to interrupt controller drivers. These routines are available from **vxbIntCtlrLib**. The utility routines fall into one of four categories: routines used during normal operation, show routines, special purpose routines not normally needed for interrupt controller drivers, and callable macros that are useful to the interrupt controller driver.



Routines used during normal operation are:

- **intCtlrHwConfGet( )**
- **intCtlrISRAdd( )**
- **intCtlrISRDisable( )**
- **intCtlrISREnable( )**
- **intCtlrISRRemove( )**
- **intCtlrPinFind( )**
- **intCtlrTableArgGet( )**
- **intCtlrTableFlagsGet( )**
- **intCtlrTableIsrGet( )**

The show routine is:

- **intCtlrHwConfShow( )**

The special purpose routines are:

- **intCtlrTableCreate( )**
- **intCtlrTableFlagsSet( )**
- **intCtlrTableUserSet( )**

The callable macros are:

- **VXB_INTCTLR_ISR_CALL( )**
- **VXB_INTCTLR_PINENTRY_ENABLED( )**
- **VXB_INTCTLR_PINENTRY_ALLOCATED( )**

The routines available to interrupt controller drivers are described in the following sections. For the prototypes, see the reference entry for the individual routines or the forward declarations in *installDir*/**vxworks-6.***x*/**target/src/hwif/intCtlr/vxbIntCtlrLib.h**.

### 4.6.1 intCtlrHwConfGet( )

**intCtlrHwConfGet( )** reads the interrupt controller resources listed in the BSP **hwconf.c** file. It follows the pointers and reads tables describing interrupt inputs, interrupt input priority, dynamic interrupt routing information (if any), and CPU configuration. When interrupt inputs are described, **isrHandle** is updated to reflect that the input is present. **isrHandle** also contains information about the input. For more information on **isrHandle**, see *4.12 Internal Representation of Interrupt Inputs*, p.65.

This routine should be called once, early in the phase 1 initialization routine, and not called subsequently.

### 4.6.2 intCtlrISRAdd( )

**intCtlrISRAdd( )** is called when a service driver connects an ISR to its interrupt. To start this process, the service driver makes a call to **vxbIntConnect( )** or **vxbDynaIntConnect( )**. Eventually, the interrupt controller driver's connect routine is called. From within its connect routine, the interrupt controller driver must take care of any required interrupt controller hardware management, and call **intCtlrISRAdd( )** to update **isrHandle** and to install the service driver's ISR.

### 4.6.3 intCtlrISRDisable( )

**intCtlrISRDisable( )** is called when a service driver disables its ISR. The interrupt controller driver must keep the interrupt input enabled if any service device ISR is enabled, and only disable the input if all ISRs connected to the interrupt input are disabled. The interrupt controller disable routine must call this routine to disable the ISR in **isrHandle** and save the return value. If the return value is **TRUE**, all ISRs on the input are disabled, and the interrupt controller can disable the interrupt input.

### 4.6.4 intCtlrISREnable( )

**intCtlrISREnable( )** is called when a service driver enables its ISR. This routine updates **isrHandle**, which results in the specified ISR being called when interrupts occur on the interrupt input.

### 4.6.5 intCtlrISRRemove( )

**intCtlrISRRemove( )** removes the specified ISR from **isrHandle**.

### 4.6.6 intCtlrPinFind( )



**intCtlrPinFind( )** is used to find the interrupt input the specified service device interrupt is connected to. The interrupt input can then be used as an argument to the other **isrHandle** support routines, and to update any tables the interrupt controller driver keeps outside of **isrHandle**. This routine is typically called once at the beginning of each routine that requires the interrupt input number, such as the routines to connect, disconnect, enable, and disable an ISR.

### 4.6.7 intCtlrTableArgGet( )

**intCtlrTableArgGet( )** retrieves the argument to the ISR for a given interrupt input. Most interrupt controller drivers do not need to call this routine. However, it is available for drivers that need to perform some action, such as moving an entire interrupt from one place to another.

### 4.6.8 intCtlrTableFlagsGet( )

**intCtlrTableFlagsGet( )** retrieves the flags for a given interrupt input. Most interrupt controller drivers do not need to call this routine.

### 4.6.9 intCtlrTableIsrGet( )

**intCtlrTableIsrGet( )** retrieves the ISR function pointer for a given interrupt input. The value returned by **intCtlrTableIsrGet( )** is a function pointer, which can contain one of three values: **intCtlrStrayISR( )**, **intCtlrChainISR( )**, or a user ISR. Most interrupt controller drivers do not need to call this routine.

### 4.6.10 intCtlrHwConfShow( )

**intCtlrHwConfShow( )** prints the contents of **isrHandle**, formatted according to the verbose level specified. This routine is always available. However, if show routines are not included in the system configuration, no output is generated.

As with all VxBus capable device drivers, each interrupt controller driver can advertise the **{busDevShow}( )** driver method. If the driver is configured to do this, the **func{busDevShow}( )** routine should make a call to **intCtlrHwConfShow( )** to provide output related to **isrHandle**.

### 4.6.11 intCtlrTableCreate( )

**intCtlrTableCreate( )** ensures that a table entry exists for the specified interrupt input. Most interrupt controller drivers do not need to call this routine.

### 4.6.12 intCtlrTableFlagsSet( )

**intCtlrTableFlagsSet( )** sets the flags variable in the **isrHandle** table for the specified interrupt input. The flags field is an unsigned integer. Most of the flags fields are used by **vxbIntCtlrLib.c** or reserved for future use. However, there are two bits available to the interrupt controller driver to use for any purpose. These are **VXB_INTCTLR_SPECIFIC_1** and **VXB_INTCTLR_SPECIFIC_2**.

### 4.6.13 intCtlrTableUserSet( )

**intCtlrTableUserSet( )** fills a table entry in **isrHandle** for a specified interrupt input. This routine fills in the specified information about the device connected to the interrupt input. The routine is called from within the **vxbIntCtlrLib** routines. Most interrupt controller drivers do not need to call this routine.

### 4.6.14 VXB_INTCTLR_ISR_CALL( )

The **VXB_INTCTLR_ISR_CALL( )** macro makes the appropriate calls to ISRs connected to a specified interrupt input. If only one ISR is connected to the interrupt input, this macro calls that ISR. If several ISRs are connected to the interrupt input, the macro walks the chain and calls each enabled ISR in turn.

For typical interrupt controller drivers, this macro should be used from within the interrupt controller driver's ISR handler, which the interrupt controller connected to the upstream interrupt controller when it called **vxbIntConnect( )** for its own interrupt outputs. For special processor architecture-specific CPU interrupt controller drivers, this macro should be used for those routines connected to the architecture-specific interrupt management code.

### 4.6.15 VXB_INTCTLR_PINENTRY_ENABLED( )

The **VXB_INTCTLR_PINENTRY_ENABLED( )** macro determines whether a specific interrupt input is enabled at the top level.

When interrupts are chained, each ISR can be enabled and disabled independently. This macro does not check the individual ISRs, but only checks the top level flag. Most interrupt controller drivers do not need to use this macro.



### 4.6.16 VXB_INTCTLR_PINENTRY_ALLOCATED( )

The **VXB_INTCTLR_PINENTRY_ALLOCATED( )** macro determines whether an **isrHandle** table entry is present for a specific interrupt input. This information is useful when generating dynamic interrupt vectors.

### 4.6.17 Dispatch Routines

In addition to the utility routines list previously, there are two routines provided by **vxbIntCtlrLib** that deserve special attention. These routines are the dispatch routines that the interrupt controller driver calls to dispatch ISRs for devices that are connected to the interrupt controller device. These routines are not called directly from the interrupt controller driver. Instead, one of the routines may be called when the interrupt controller driver makes a call to **VXB_INTCTLR_ISR_CALL( )**, depending on whether or not the ISRs are attached to the interrupt input.

The routine **intCtlrStrayISR( )** is called when no ISR is attached to the interrupt input. The routine **intCtlrChainISR( )** is called when more than one ISR is attached to the interrupt input. If your driver meeds to know how many ISRs are connected to an interrupt input, the driver can call **intCtlrTableIsrGet( )**. If the value returned is **intCtlrStrayISR( )**, no ISRs are connected. If the value returned is **intCtlrChainISR( )**, more than one ISR is connected. If the value is any other non-null value, a single ISR is connected to the specified interrupt input.

In most cases, your interrupt controller driver does not need to know this information. However, in some cases, such as those dealing with dynamic vector assignment, this information can be useful.

In addition to these dispatch routines, there are routines available to help the interrupt controller driver manage dynamically assigned vectors. If the dynamic support library is included in the system configuration, these routines are available

as function pointers. The function pointers include **vxbIntDynaCtlrInputInit( )** and **vxbIntDynaConnect( )**.

#### vxbIntDynaCtlrInputInit( )

In some cases, the interrupt controller driver may wish to provide one or more separate interrupt tables for dynamic interrupt sources. The **vxbIntDynaCtlrInputInit( )** routine initializes these tables.

```
STATUS (*_func_vxbIntDynaCtlrInputInit)

    struct intCtlrHwConf *isrHandle,
    struct dynamicIntrTable *entry,
    void *dynamicIsr
```

#### vxbIntDynaVecProgram( )

When necessary, your interrupt controller driver must program dynamically generated interrupts into the devices that have dynamically generated vectors assigned to them. This is accomplished by calling **vxbIntDynaVecProgram( )**.

```
STATUS (*_func_vxbIntDynaVecProgram)
    (
    VXB_DEVICE_ID pVectorOwner,
    VXB_DEVICE_ID serviceInstance,
    struct vxbIntDynaVecInfo * pDynaVec
    )
```

## 4.7 Initialization

By the beginning of VxBus initialization phase 2, interrupt controller drivers must be able to connect ISRs at the request of other drivers. Because the phase 2 initialization routine for an interrupt controller driver may not run before other drivers attempt to connect their ISRs, interrupt controllers must do all of their initialization in phase 1.

## 4.8 Interrupt Controller Typologies and Hierarchies



Every interrupt controller has some number of interrupt inputs. The number of inputs may be one, or it may be a large number of inputs. In addition to interrupt inputs, each interrupt controller has one or more interrupt outputs. This is where interrupts are generated. Most interrupt controllers treat their interrupt outputs in the same manner that other drivers handle interrupt generation. That is, the controllers connect an ISR using **vxbIntConnect( )**. When any **vxbIntConnect( )** call is made, an interrupt controller in the system claims the interrupt. This response is the same, whether the caller to **vxbIntConnect( )** is an interrupt controller instance or an instance from some other device class. This implies that interrupt controllers have a hierarchy of connectivity, where the inputs of some interrupt controllers are connected to the outputs of other interrupt controllers. The management of each interrupt controller device is separated, because each interrupt controller is represented by a separate VxBus instance.

Within this hierarchy, each CPU can be considered to be an interrupt controller device at the top of the interrupt controller device tree. CPU interrupt controller devices are special in a number of ways. Although these drivers handle interrupt inputs in a manner similar to other drivers, they handle interrupt outputs in a special manner. The drivers do not try to connect interrupt outputs using the VxBus interrupt connection mechanism. Instead, they connect to the architecture-specific code that is provided for interrupt connection.

Interrupts can be delivered as messages rather than values on a physical wire. This may be the case for interrupt handling on the CPU. It can also be the case when an external bus controller and interrupt controller are included on the same device, such as with the PCI-X and PCIe bus controller devices used on some PowerPC processors. Typically, there are several things that the interrupt controller instance must do differently when interrupts are delivered as messages versus when they are hard-wired interrupts. This can include assignment of a vector (which in this context is simply an identification number) for each device that generates interrupt messages.

## 4.9 Interrupt Priority

Within the VxBus interrupt controller design, each interrupt input can be assigned a software priority value. The priority is represented as a 32-bit unsigned integer, which allows a larger range of interrupt priorities than any existing hardware provides. Each driver needs to map the priority ranges available in hardware to the range allowed for software.

The highest software priority value is zero. Where the hardware supports different priority levels, the hardware priority level of any software priority level must be equal to or less than the hardware priority level of the next higher software priority number, as follows:

```
hwPrio(swPrio(N)) <= hwPrio(swPrio(N-1))
```

Table 4-1 shows the possible mappings between hardware priority and software priority for a an example where the given piece of hardware provides 32 hardware priority levels and 0 is the highest priority. In the table, *N* is some starting point determined by a device parameter.

Table 4-1 **Hardware and Software Priority Mappings**

| Hardware Priority | Software Priority |
|---|---|
| 0 | *N* through *N*+3 |
| 1 | *N*+4 through *N*+7 |
| 2 | *N*+8 through *N*+11 |
| ... | ... |
| 31 | *N*+256 through 0xffffffff |

There are many reasonable priority mapping schemes, and an interleave of 4, as shown in Table 4-1 is only one valid scheme. The only important consideration is that each software priority level be mapped to a hardware priority level with the same priority or greater priority than each lower-numbered software priority level.

In some cases, the hardware has a fixed hardware priority scheme (for example, the I8259 interrupt controller device).When there is a fixed hardware priority scheme, the only software priority that can be assigned is the priority of the first interrupt input. All other interrupt input priority levels are determined by the first interrupt input. However, mapping between hardware and software priority

levels must still be performed, because the user may perform some actions on devices with specific interrupt priority levels.

Where software assigned priority is available, the default priority must be set at 15. Due to special considerations on some hardware, priority levels of 0 and 1 should never be used for external devices.



## 4.10 ISR Dispatch

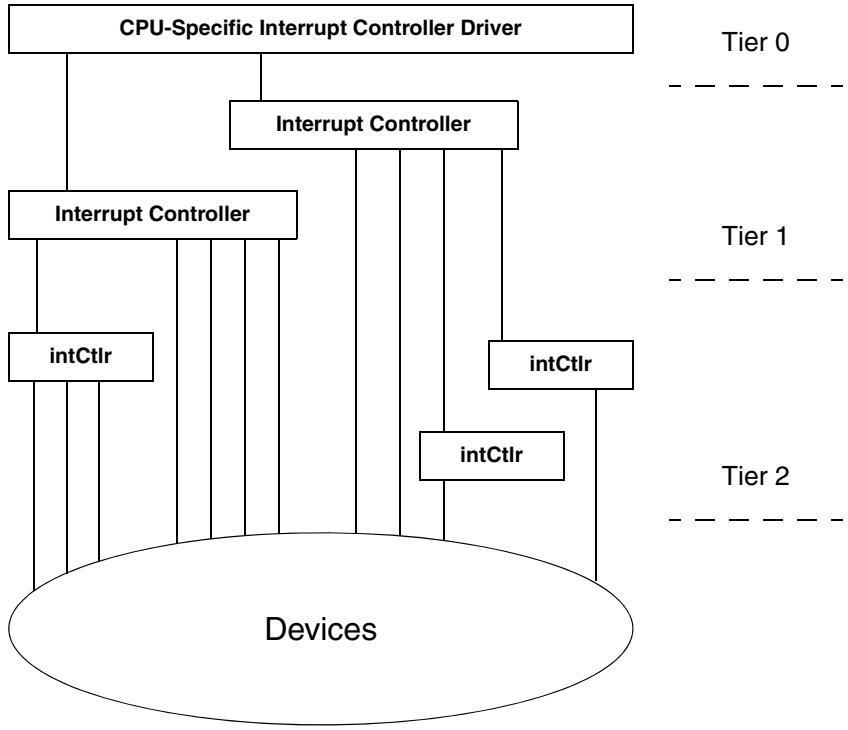
In order to understand how ISR dispatch works, you must understand the interrupt controller layers involved. This section discusses these layers, how they interact, and the terminology associated with them. It also discusses the ISR dispatch process itself.

In this section, the CPU-specific driver is referred to as tier 0, the interrupt controller driver(s) connected directly to the CPU-specific interrupt controller are referred to as tier 1, interrupt controllers connected to tier 1 are referred to as tier 2, and so on.

Tier 0 interrupt controller drivers are always architecture-dependent or CPU-dependent. Devices used as tier 1 interrupt controllers are typically, though not necessarily, used only on a single processor architecture. Devices used as tier 2 and lower interrupt controllers are not typically architecture specific.

Figure 4-1 illustrates this layering.

Figure 4-1 **Interrupt Controller Tier Mapping**




Every interrupt controller driver, regardless of the tier on which it resides, dispatches downstream ISRs by invoking the **VXB_INTCTLR_ISR_CALL( )** macro. In general[1], the tier 1 interrupt controller ISR must:

1. Mask off interrupts from the source that generated the interrupt.
2. Re-enable interrupts with a call to **intCpuUnlock( )**.
3. Invoke **VXB_INTCTLR_ISR_CALL( )**.
4. Disable interrupts with a call to **intCpuLock( )**.

Because there is no previous **intCpuLock( )** call to return the appropriate lock value, it is difficult for the system to determine what argument to use for

1. Where the architecture already re-enables interrupts of higher priority, the interrupt controller driver does not need to do so. This currently happens on the MIPS architecture only.

**intCpuUnlock( )**. Because tier 1 interrupt controller devices are typically used with only a single architecture, interrupt controller drivers for those devices can use architecture-specific information for the argument to **intCpuUnlock( )**. However, for good programming practice, the value of the argument should be available as a macro that can be set differently according to the CPU macro, and an error generated if the driver is compiled for any unsupported architecture. For example:

```
#if CPU==PPC32 IMPORT int vxPpcIntMask;
#define SAMPLE_INTCTLR_INTMASK vxPpcIntMask
#else /* CPU==PPC32 */
#error vxbSampleIntCtlr not available for this architecture
#endif /* CPU==PPC32 */
```

In accordance with the design goal of minimizing the number of interrupts that occur, interrupt controller drivers should process all pending unmasked interrupts whenever the interrupt controller driver ISR is called. Often, this means that the driver reads a register to determine which inputs have pending interrupts, and processes each interrupt source in a loop.

The following example is modified from the EPIC interrupt controller driver:

```
/* lock interrupts and find key */

key = intCpuLock();

/* start with input 0 */

inputNo = 0;

/* find pending interrupts */

pendSet = vxbRead32(...);

while ( pendSet != 0 )
    {

    if ( pendSet & 1 )
        {
        /* pending: dispatch downstream ISRs */

        intCpuUnlock(vxPpcIntMask);
        VXB_INTCTLR_ISR_CALL(isrHandle, i);
        dontCare = intCpuLock();
        }

    /* find next input and adjust pendSet */

    inputNo++;
    pendSet >>= 1;
    }

intCpuUnlock(key);
```

By using the **intCpuUnlock( )** and **intCpuLock( )** calls in the tier 1 interrupt controller ISR, the system provides priority dispatching of interrupts for devices connected directly to tier 1 interrupt controller devices. Currently, VxWorks does not provide a mechanism to perform similar priority dispatching at other tiers.

If your application requires priority dispatching at other tiers, see the Wind River Online Support Web site for supplemental documentation and for the most recent interrupt controller drivers.

## 4.11 Managing Dynamic Interrupt Vectors

Some bus types allow dynamic assignment of interrupt values, often referred to as vectors. For example, variants of PCI bus may provide message signalled interrupts (MSI), which require firmware or software to assign the vector. There is also an MSI-X variant, which is a different representation of dynamic interrupt vectors on variants of the PCI bus type.

In addition, some interrupt controller devices reside on multifunction chips. Multifunction chips can include an interrupt controller device in addition to other devices, and may have a register containing the interrupt vector to use when the device generates an interrupt. The driver software can, and often must, write a valid vector to this register in order for the device to generate an interrupt. And the vector used must be generated somehow, possibly dynamically at runtime.

The VxBus interrupt controller driver design provides the ability for interrupt controller drivers to manage dynamically generated interrupts.

There are two ways that a dynamically generated vector can be assigned to a specific device. The first method is used when the driver makes a call to a special API for connecting ISRs to dynamically generated interrupts. The second method is used when a BSP is configured with devices connected to **VXB_INTR_DYNAMIC** as described in *4.5.2 Dynamic Vector Table*, p.48.

### Configuring Dynamic Vectors Using the Service Driver Routines

The service driver can call **vxbIntDynaConnect( )** to connect an ISR to a dynamically assigned interrupt. For clarity, the **vxbMsiConnect( )** alias is available for MSI on PCI bus. These routines allow the service driver to provide a list of ISRs and arguments to connect to multiple dynamically assigned interrupts.

The **vxbIntDynaConnect( )** routine can be used when a driver configures the device to use multiple interrupts, where the bus type otherwise prevents multiple interrupts from being used. For example, normal PCI bus operation requires that a single interrupt be used for each function on a PCI card. In a network device, all interrupt types (transmit, receive, and error) share the same interrupt. To increase performance, your driver can split transmit, receive, and error interrupts into separate interrupts and provide a customized ISR for each interrupt type. This reduces the overhead of checking whether each type of condition occurs. That is, when only the transmit interrupt is active, the driver does not need to check for receive conditions or error conditions.

When **vxbIntDynaConnect( )** is called, the dynamic interrupt library finds an interrupt controller that publishes the **{vxbIntDynaVecConnect}( )** driver method. An internal routine then calls **func{vxbIntDynaVecConnect}( )** for the interrupt controller that responded. This routine must assign vectors to use for the device, connect the ISRs provided by the driver, configure the interrupt controller hardware to accept the newly assigned vectors, and program the vectors into the requesting service device as described in *Programming Dynamic Vectors*, p.64.

**NOTE:** The interrupt controller driver is responsible for programming the dynamic vectors into the device.

### Configuring Dynamic Vectors in the BSP

The BSP can configure any device to be connected to the interrupt controller **VXB_INTR_DYNAMIC** input. When this is the case, the service driver calls **vxbIntConnect( )** to connect a single ISR as usual. The **vxbIntConnect( )** routine follows the normal procedure to identify the interrupt controller to which the device is connected and calls the **func{vxbIntCtlrConnect}( )** provided by the interrupt controller driver.

In order to support BSP configuration of dynamic vectors, the **func{vxbIntCtlrConnect}( )** in the interrupt controller driver must find the input pin to which the device is connected, using **intCtlrPinFind( )**. It must then check the value returned by **intCtlrPinFind( )** to see if the value is **VXB_INTR_DYNAMIC**.

If so, the routine follows the same procedure it does when the **func{vxbIntDynaVecConnect}( )** routine is called, That is, it assign vectors to use for the device, connects the ISRs provided by the driver, configures the interrupt controller hardware to accept the newly assigned vectors, and programs the vectors into the requesting service device as described in *Programming Dynamic Vectors*, p.64.

**NOTE:** The interrupt controller is responsible for programming the dynamic vectors into the device.

### Programming Dynamic Vectors

The last stage of dynamic vector installation is to program the dynamic vector into the device. The interrupt controller is responsible for initiating this process, but the interrupt controller is not expected to know how to do so. Instead, one of two entities on the system must know how to program the dynamic vectors into the device. Those two entities are the service device itself, and the bus controller immediately upstream from the device. One or both of these entities must indicate that they know how to program the dynamic vector into the device by publishing the **{vxbIntDynaVecProgram}( )** driver method. PCI bus controller drivers normally publish this method. However, because the code to program the vectors into an MSI-capable device is independent of the bus controller, the PCI library provides the routine **vxbPciMSIProgram( )** to perform the actions. When the bus type does not support dynamic vectors, the device itself must provide a custom routine to program the dynamic vector.

The interrupt controller can perform the actions to check for the **{vxbIntDynaVecProgram}( )** driver method and call it, by simply calling through the function pointer **_func_vxbIntDynaVecProgram**:

```
if ( _func_vxbIntDynaVecProgram != NULL )
    {
    (*_func_vxbIntDynaVecProgram)(devID, dynaVec);
    }
```

The prototypes for the driver method and **_func_vxbIntDynaVecProgram** are as follows:

```
STATUS func{vxbIntDynaVecProgram}
    (
    VXB_DEVICE_ID pInst,
    struct vxbIntDynaVecInfo *dynaVec
    )
```

```
STATUS _func_vxbIntDynaVecProgram
    (
    VXB_DEVICE_ID pInst,
    struct vxbIntDynaVecInfo *dynaVec
    )
```



#### Determining Dynamic Vector Values

The interrupt controller driver must choose the dynamic vector according to constraints in the hardware. Within this range, there are several things to keep in mind.

The best system performance is obtained when ISRs are not chained. Therefore, dynamically assigned vectors should be unassigned to other devices whenever possible.

When multiple dynamically assigned vectors are available, they should be sequential. The interrupt controller driver may be able to scatter multiple dynamically assigned vectors throughout the range of acceptable vectors, but VxWorks does not support this functionality.

## 4.12 Internal Representation of Interrupt Inputs

When interrupt controller drivers use **vxbIntCtlrLib** functionality, the interrupt inputs must be represented by the structures used by **vxbIntCtlrLib**. The data are kept in a structure, referred to as the **isrHandle**, which contains information about all interrupt inputs and the ISRs that are connected to them.

The information kept in the **isrHandle** includes a two tier system, where the lower tier consists of an array of structures, each containing information about a single interrupt input. Each entry in this array is referred to as an interrupt input table entry. The upper array consists of a simple pointer to the first element of the second tier array.

In order to improve memory efficiency for the most common interrupt controllers, the current implementation limits the low level table to eight inputs. In order to be able to support the maximum number of inputs, the top level table size is 496. These values may change in a future release, therefore the macros **VXB_INTCTLRLIB_LOWLVL_SIZE** and **VXB_INTCTLRLIB_TOPLVL_SIZE** should be

used whenever the code needs to know the maximum number of interrupt inputs that can be represented.

**NOTE:** The table sizes listed in this section represent the sizes used at the time of publication and are subject to change. For the current sizes, refer to the values of **VXB_INTCTLRLIB_TOPLVL_SIZE** and **VXB_INTCTLRLIB_LOWLVL_SIZE** defined in *installDir*/**vxworks-6.***x*/**target/src/hwif/intCtlr/vxbIntCtlrLib.h**.

The current values for these macros result in the ability for each table to represent up to 3968 interrupt inputs. If you are working with an interrupt controller that has more 3968 inputs, you can choose one of two options.

Where possible, you should limit the number of supported inputs to a value less than the value described by the following formula:

```
( VXB_INTCTLRLIB_TOPLVL_SIZE * VXB_INTCTLRLIB_LOWLVL_SIZE ).
```

This may be possible for interrupt controllers that use a small number of hard-wired inputs, and also allow for many dynamically assigned interrupts. In this case, you can simply choose to not support the full range of dynamically assigned interrupts supported by the hardware.

When the driver needs to support all of the inputs provided by the hardware, you can choose to represent the inputs in more than one input table. The utility routines in **vxbIntCtlrLib** support this option, because they require the input table as an argument, rather than some other structure. However, adding this support in your interrupt controller driver is more complex and may, in some cases, result in slower interrupt performance.

## 4.13 Multiprocessor Issues with VxWorks SMP

A multiprocessor (MP) system is a computer system with more than one processor. There are several common configurations of MP systems.

The most common multiprocessing configuration is referred to as asymmetric multiprocessing (AMP). There are two variations of this. In one configuration, there are different kinds of processors on the system, possibly with different instruction sets. Typically, one processor is considered the master system, and other processors perform dedicated assignments.

The second AMP configuration includes some number of identical processors with each processor running a separate OS or a separate invocation of the same OS.

There is a third option for systems with multiple identical processors. When all of the processors in a system are identical, and a single OS is running on all processors at the same time, the system is called a symmetric multiprocessing (SMP) system.

There are some aspects of MP systems that require special handling from the interrupt controller driver. This section describes those special MP considerations.



### 4.13.1 Routing Interrupt Inputs to Individual CPUs

With the optional VxWorks SMP product, individual interrupts can be routed to processors other than the boot processor. However, the system requires that peripheral devices be initialized before additional processors are brought online. For this reason, when VxWorks SMP boots, all interrupts are initially routed to the boot processor, and a sequence of events is used to reroute interrupts from the boot processor to other processors.

The additional processors are brought online with a call to **usrEnableCpu( )**. This routine iterates through the additional processors and enables each in turn. As each processor is brought online, the system reroutes all interrupts destined for that processor to it with a call to **vxbIntToCpuRoute( )**. This routine walks the list of devices and runs the **{vxbIntCtlrCpuReroute}( )** method for each one.

```
STATUS func{vxbIntCtlrCpuReroute}
    (
    VXB_DEVICE_ID       pInst,
    void *              destCpu
    )
```

When an interrupt controller driver's **func{vxbIntCtlrCpuReroute}( )** routine is called, this routine needs to check each interrupt input to see whether it is configured for the specified destination CPU, **destCpu**. If so, it configures the hardware so that **destCpu** receives all interrupts that arrive on that interrupt input. The **VXB_INTCTLR_PINENTRY_ALLOCATED( )** macro and the **vxbIntCtlrPinEntryGet( )** routine are useful for accomplishing this task. The driver first checks whether any entry is allocated for the specified interrupt input. If an entry is allocated, the driver finds the table entry with **vxbIntCtlrPinEntryGet( )** and reads the **pinCpu** field of the table structure. If the **pinCpu** field matches the **destCpu** argument, then that interrupt input is rerouted to **destCpu**.

The interrupt controller driver is not finished at this point. Some service drivers defer servicing the device interrupt in the ISR. Instead of performing all of the operations that are required when the interrupt occurs, these drivers simply configure the device so that it does not generate interrupts, and then enable a task to run, which services the interrupt.

**NOTE:** The **isrDeferLib** routines can be used to assist with this situation, but other mechanisms are possible. For the purpose of this discussion, the service driver uses **isrDeferLib**.

When the interrupt input is rerouted to **destCpu**, the actual interrupt may be processed on **destCpu**, but the defer task can be running on a different CPU. This is unlikely to result in the intended system performance, therefore each service driver with an ISR connected to the interrupt input should be instructed to set an appropriate CPU affinity for the defer task.

The interrupt controller driver makes a call to **isrRerouteNotify( )**. This routine walks the chain of ISRs connected to the interrupt input, and checks each connected instance for the **{isrRerouteNotify}( )** driver method. If the instance publishes this method, the **func{isrRerouteNotify}( )** routine is called. The **func{isrDeferIsrRerouteNotify}( )** routine must perform whatever device-specific operations that are required to accommodate the rerouting of its interrupt to a different CPU. For example, if the driver uses the **isrDeferLib** library, it calls that library's **isrDeferIsrReroute( )** routine to announce the rerouting of its interrupt to the library[2].

### 4.13.2 Interprocessor Interrupts

On multiprocessor systems, there are several OS modules that must be able to interrupt individual processors on the system. The OS modules that require this functionality include the scheduler (and any other module that manages tasks), the OS debug support module, and, potentially, every module that requires management of cache and MMU. The mechanism to interrupt individual processors is called interprocessor interrupts, or IPIs.

2. For additional information about the use of **isrDeferLib** in an SMP environment, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.

**NOTE:** What the kernel modules do when they invoke IPIs is beyond the scope of this document. This document focuses only on generating IPIs, which is the responsibility of interrupt controller drivers. For more information, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.



In the optional VxWorks SMP product, kernel debug support modules use **vxIpiLib** to generate IPIs. The routines in **vxIpiLib** rely on VxBus interrupt controller drivers to perform the actual work. Other kernel modules, such as the scheduler, use an internal library to perform inter-processor interactions, which resolve to **vxIpiLib** calls. In all of these cases, the system ends up calling a routine provided by an interrupt controller driver in order to generate the IPI.

The mechanism used to support IPIs relies on a single driver method, **{vxIpiControlGet}( )**. This method returns a pointer to a structure that describes the kinds of IPIs that the interrupt controller driver can generate. The structure contains:

- several function pointers that are called to perform various operations related to IPIs
- a list of CPUs that this interrupt controller device can interrupt
- a count of the number of different IPIs that this interrupt controller can generate

The structure is the **VXIPI_CTRL_INIT** structure, which is defined in *installDir***/vxworks-6.***x***/target/h/vxIpiLib.h** as follows:

```
typedef struct vxIpiCntrlInit
    {
    SL_NODE                 ipiList;        /* Next IPI structure          */
    cpuset_t                pCpus;          /* destination CPUs            */
    VXIPI_EMIT_FUNC         ipiEmitFunc;    /* Trigger an IPI int          */
    VXIPI_CONNECT_FUNC      ipiConnectFunc; /* Install an IPI int handler  */
    VXIPI_ENABLE_FUNC       ipiEnableFunc;  /* Enable int                  */
    VXIPI_DISABLE_FUNC      ipiDisableFunc; /* Disable int                 */
    VXIPI_DISCONN_FUNC      ipiDisconnFunc; /* Disconnect handler          */
    VXIPI_PRIOGET_FUNC      ipiPrioGetFunc; /* Get IPI priority            */
    VXIPI_PRIOSET_FUNC      ipiPrioSetFunc; /* Set IPI priority            */
    INT32                   ipiCount;       /* Number of IPIs available    */
    VXB_DEVICE_ID           pCtlr;          /* Interrupt Controller        */
    } VXIPI_CTRL_INIT, * VXIPI_CTRL_INIT_PTR;

VXIPI_CTRL_INIT * func{vxIpiControlGet}
    (
    VXB_DEVICE_ID pInst,
    void * ignored
    )
```

The routines pointed to by the **VXIPI_CTRL_INIT** structure function pointers—which the interrupt controller driver must provide—have the following prototypes:

```
/*******************************************************************************
*
* ipiGen - Generate Inter Processor Interrupt
*
* This functions generates a IPI interrupt at the target CPU sets specified
* by the second argument. The first arguments can be of the four IPI channels
* available at the EPIC.
*/

LOCAL STATUS ipiGen
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId,
    cpuset_t cpus
    )

/*******************************************************************************
*
* ipiConnect - Connect ISR to IPI
*
* This routine connects the specified ISR and argument to the IPI specified
* by the ipiId argument.  The pCtlr argument refers to the interrupt
* controller.
*/

LOCAL STATUS ipiConnect
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId,
    IPI_HANDLER_FUNC ipiHandler,
    void * ipiArg
    )

/*******************************************************************************
*
* ipiEnable - Enable specified IPI
*
* This routine enables generation of the IPI specified by the ipiId argument.
*
*/

LOCAL STATUS ipiEnable
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId
    )
```

```
/*******************************************************************************
*
* ipiDisable - Disable specified IPI
*
* This routine disables the IPI specified by the ipiId argument.
*
*/

LOCAL STATUS ipiDisable
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId
    )

/*******************************************************************************
*
* ipiDisconn - Disconnect ISR from IPI
*
* This routine disconnects the specified ISR and argument from the IPI
* specified by the ipiId argument.  The pCtlr argument refers to the
* interrupt controller.
*
*/

LOCAL STATUS ipiDisconn
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId,
    IPI_HANDLER_FUNC ipiHandler,
    void * ipiArg
    )

/*******************************************************************************
*
* ipiPrioGet - Retrieve IPI priority
*
* This routine returns the interrupt priority of the IPI specified by the
* ipiId argument.  Note that the priority is a software priority, which
* may not correspond directly to hardware priority.
*
*/

LOCAL INT32 ipiPrioGet
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId
    )
```

```
/*******************************************************************************
*
* ipiPrioSet - Set IPI priority
*
* This routine changes the interrupt priority of the IPI specified by the
* ipiId argument to the value specified by the prio argument.  Note that the
* priority is a software priority, which may not correspond directly to
* hardware priority.
*/

LOCAL STATUS ipiPrioSet
    (
    VXB_DEVICE_ID pCtlr,
    INT32 ipiId,
    INT32 prio
    )
```

Within the VxBus interrupt controller design, IPIs are represented outside the interrupt controller driver by a simple integer value. This value is an index of the IPI. The value may reflect some vector information. However, any relationship between the IPI ID and any vector should be hidden in the interrupt controller driver. An interrupt controller device that can generate eight distinct IPIs has **ipiID** values ranging from zero to seven.

Depending on the system configuration, one or more **ipiID** values are reserved for system use. The remainder may be available for application use. **ipiID** 1 is always reserved for debug support, and is not available to applications. When the optional VxWorks SMP product is used, **ipiID** 0 is reserved for CPC calls used by the OS, and all remaining IPIs are reserved and therefore not available for application use.

In addition, there may be special considerations on some BSP or hardware platforms that require the interrupt controller to reserve additional interrupts for other purposes. These interrupts are an exception to the reserved interrupts for VxWorks SMP. This situation is rare and should not be required in most cases.

### 4.13.3 Limitations in Multiprocessor Systems

For certain limitations in multiprocessor systems, interrupt controllers may be required to assist in a workaround. The category of limitation described here contains those issues related to the use of SMP on hardware that is not truly symmetric. That is, the system includes some devices that cannot be managed equally by all processors.

As stated previously, the system is configured and all peripheral devices are initialized before any additional processors are brought online. However, in some systems there are devices that the boot processor does not have access to. These devices cannot be brought online until after some other processor is brought up. In

order to support these devices, a special BSP configuration may be used to allow the devices to be started. However, when the ISRs for the devices are connected, the interrupt controller must be able to route the ISRs to one of the processors to which the device is connected. If VxWorks SMP must be used on this type of asymmetric hardware platform, the interrupt controller can choose to have special-purpose resources provided by the BSP to indicate restrictions of this nature.

A similar situation can occur for devices connected directly to the boot processor and unavailable to other processors. However, in this situation, the problem is reversed. The system works fine as long as those devices are not rerouted to other processors. In this case, the best solution is to ignore the issue. If applications attempt to reroute those devices to processors that do not have access to the device registers, the device simply fails. Application developers should consider this situation during their development.

## 4.14 Debugging

Interrupt controller drivers are one of the most difficult driver classes to debug. Because the serial console and network interfaces are not available until the interrupt controller driver is available, it is not normally possible to defer driver registration. Therefore, it is not possible to establish a debug session with a working VxWorks system until after the interrupt controller driver is working correctly.

The recommended debugging mechanism for interrupt controller drivers is to use a hardware debugger. When the hardware debugger is combined with a suitable graphical interface that includes knowledge of source code, interrupt controller drivers can be debugged more efficiently.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 5 Multifunction Drivers


5.1 Introduction 75

5.2 Overview 76

5.3 VxBus Driver Methods 76

5.4 Header Files 77

5.5 BSP Configuration 77

5.6 Available Utility Routines 78

5.7 Initialization 79

5.8 Device Interconnections 79

5.9 Logical Location of Subordinate Devices 81

5.10 Debugging 82


## 5.1 Introduction

This chapter describes multifunction drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 5.2 Overview

One trend in hardware design is to combine more and more devices onto a single chip. This has 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. Application developers should be able to exclude features that are not needed by their application, including device support. Therefore, rather than write a driver for a complete ASIC, you should instead write your drivers as though each subsection is a separate device.

The VxBus framework assists in this process by allowing you to create a driver for each component on the chip, and then provide a single multifunction driver for the entire chip. The purpose of the single multifunction driver is to inform VxWorks and VxBus of the presence of the different devices on the chip so that they can be individually matched with a driver. Using a multifunction driver significantly reduces the complexity of your BSP configuration. In most cases, the configuration can be simplified such that it provides only a single **hwconf.c** file entry for the entire chip.

Subdivision of the device registers and creation of subordinate devices is also handled by the multifunction driver.

## 5.3 VxBus Driver Methods

Multifunction drivers do not use or supply any VxBus driver methods. During VxBus initialization, the driver initializes the multifunction chip (if required) and announces the devices on the chip to VxBus.

## 5.4 Header Files

There are no custom header files available for use with multifunction drivers. However, in some ways, the functionality provided by multifunction drivers is similar to that provided by bus controller drivers. For this reason, multifunction drivers must include **vxBus.h** in order to create device structures for the individual devices on the chip. For example:

```
#include <hwif/vxbus/vxBus.h>
```



## 5.5 BSP Configuration

Multifunction drivers do not typically require configuration information from a BSP that is above and beyond the normal device-specific information provided for all drivers. One exception is when not all devices available on the chip are supported by a driver in the system (see *Limited Device Support in the Driver*, p.77).

For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.

### Limited Device Support in the Driver

Although most multifunction devices require no class-specific BSP configuration steps, there is one possible exception. If your target system is configured with drivers for only one or two of the devices on the chip, your multifunction driver can choose not to inform VxBus of devices for which no driver is present. In this case, the data space for the device structures is not allocated leading to a smaller footprint. However, the benefit of reduced footprint is often outweighed by the increased size and complexity of the multifunction driver. This configuration also requires that the driver be compiled at VxWorks build time. For these reasons, Wind River does not recommend using this configuration.

## 5.6 Available Utility Routines

The primary purpose of a multifunction driver is to allocate the required device structures and fill in the data fields of each device structure. The available utility routines for a multifunction driver is discussed in this section. For more information on these routines, see the reference entries for **vxBus.c**.

#### vxbDevStructAlloc( )

The prototype for **vxbDevStructAlloc( )** is as follows:

```
VXB_DEVICE_ID vxbDevStructAlloc( )
```

This routine allocates a device structure.

#### vxbDeviceAnnounce( )

The prototype for **vxbDeviceAnnounce( )** is as follows:

```
STATUS vxbDeviceAnnounce(VXB_DEVICE_ID devID)
```

This routine announces a new device to VxWorks and VxBus. The device structure must already be allocated and the data filled in.

#### vxbDevRemovalAnnounce( )

The prototype for **vxbDevRemovalAnnounce( )** is as follows:

```
STATUS vxbDevRemovalAnnounce(VXB_DEVICE_ID devID)
```

This routine informs VxWorks and VxBus that a device is being removed from the system.

#### vxbDevStructFree( )

The prototype for **vxbDevStructFree( )** is as follows:

```
void vxbDevStructFree(VXB_DEVICE_ID devID)
```

This routine returns a device structure to the pool, making it available for future device allocation.

**vxbBusAnnounce( )**

The prototype for **vxbBusAnnounce( )** is as follows:

```
STATUS vxbBusAnnounce
    (
    struct vxbDev *     pBusDev,    /* bus controller */
    UINT32              busID       /* bus type */
    )
```

The **vxbBusAnnounce( )** routine is used to create a new bus —subordinate to the multifunction device—on which any downstream devices reside.



## 5.7 Initialization

There are no class-specific initialization restrictions on multifunction drivers. However, subordinate devices should be announced to VxBus as early in the initialization process as possible.

## 5.8 Device Interconnections

Within multifunction devices, it is common for there to be interactions between the subordinate devices. This usually takes one of two forms—interleaved registers or shared resources— but other kinds of interactions are also possible. This section addresses these interaction types.

### 5.8.1 Interleaved Registers

In some multifunction chips, registers for individual device parts are interleaved in the address space assigned to the chip. For example, there may be registers located at base+0x00000000 through base+0x0000ffe0, additional registers located at base+0x00010040 through base+0x000100ff, and other registers scattered through base+0x00020000 to base+0x0003ffff. Your multifunction driver must handle this condition.

Interleaved registers can be supported by the driver in two ways. The first method you can use to handle this condition in your driver is to provide register access routines that remap the registers of the subordinate devices so that they look like a single bank of registers. This method results in slower performance due to longer time to access device registers. However, when one or more subordinate devices use pre-existing drivers that assume a single uniform register block, this is the preferred mechanism. Otherwise, use the second method.

The second method you can employ in your driver to handle the condition of interleaved registers for subordinate devices requires cooperation with the drivers for the affected subordinate devices. For this method, the multifunction driver can choose to define small banks of specific registers for each subordinate device.

There are ten register base addresses available to VxBus drivers. (For more information on register access, see the hardware access section of *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.) Using the example described earlier in this section, the multifunction driver can assign the subordinate device register bases as follows:

| Base | Address | Size |
|---|---|---|
| **pRegBase[0]** | base+0x00000000 | 0x0000ffe0 / 65504 |
| **pRegBase[1]** | base+0x000207e0 | 0x00000020 / 32 |
| **pRegBase[2]** | base+0x00010040 | 0x000000c0 / 192 |
| **pRegBase[3]** | base+0x00020914 | 0x00000004 / 4 |
| **pRegBase[4]** | base+0x00023ff0 | 0x00000010 / 16 |

The multifunction driver and the drivers for the individual devices must agree regarding which **pRegBase[ ]** entry to use for each register, as well as the offset.

To achieve this agreement, you can assign the appropriate values to the **pRegBase[ ]** entries in the structure. Using the previous example, the code might look similar to the following:

```
devID = vxbDevStructAlloc()
...
base = myDevID->pRegBase[0] + CURRENT_DEVICE_OFFSET;
devID->pRegBase[0] = base;
devID->pRegBase[1] = base + 0x207e0;
devID->pRegBase[2] = base + 0x10040;
devID->pRegBase[3] = base + 0x20914;
devID->pRegBase[4] = base + 0x23ff0;
```

**NOTE:** Wind River strongly recommends that multifunction drivers do not simply make the entire register bank available to all subordinate drivers. This increases the probability of a condition where a bug in one driver can result in symptoms that show up in another driver. This is difficult to debug.



### 5.8.2 Shared Resources

When multiple devices on a single multifunction chip share a set of resources also available on the same chip, you may find it useful to create a driver to manage those resources. This is called a *resource driver* (see *8. Resource Drivers*). This driver can simply allocate a resource to one of the other drivers, assuming that the other driver knows how to make use of the resource, or it can provide an API to manage the resource on behalf of the user. If a resource driver is used, the multifunction driver should be configured so that it requires the resource driver to be present in the system.

### 5.8.3 Other Interactions

In some cases, hardware designs require interactions among the subordinate devices on a multifunction chip that do not fall into either of the categories described previously. These interactions are varied, and therefore difficult to describe in a general discussion. These interactions can include reduced or increased functionality for the multifunction version of the device compared against non-multifunction versions of the device, or there may be hardware bugs due to unforeseen interactions of the component parts of a multifunction chip. In all cases, the interactions must be handled as appropriate for the chip, in whichever driver or drivers are appropriate.

## 5.9 Logical Location of Subordinate Devices

You can write your multifunction driver in such a way that the devices are seen as located either on the parent bus of the multifunction device, or on a bus subordinate to the multifunction device. If you choose a subordinate bus, it can be either a multifunction bus type or the same type as the parent bus.

Drivers written for subordinate devices should be written to accept devices on either a multifunction bus or on the upstream bus type such as PLB or PCI. If you need to use pre-existing drivers that do not provide this flexibility and cannot be modified, your multifunction driver may be forced to locate subordinate devices on the upstream bus.

## 5.10 Debugging

Typically, multifunction drivers can be debugged easily after the system is booted. Simply download the driver object module and run the registration routine. Use **vxBusShow( )** to see whether the downstream devices show up as instances or as orphans.

Custom drivers for subordinate parts of a multifunction chip are debugged based on the driver class to which they belong.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 6 Network Drivers


6.1 Introduction 83

6.2 Network Interface Drivers 86

6.3 PHY Drivers 123

6.4 Wireless Ethernet Drivers 133

6.5 Hierarchical END Drivers 134


## 6.1 Introduction

This chapter describes several types of VxWorks network drivers. This chapter includes the primary documentation for network interface drivers (also known as VxbEnd drivers or MAC drivers) and PHY drivers. It also includes a brief overview and pointer to additional information for Wind River Wireless Ethernet Drivers. The final section briefly discusses hierarchical END drivers, which are deprecated.

### 6.1.1 Terminology

Media access controller (MAC) devices are commonly thought of as network interfaces. In this document, the term *media access controller* and the acronym *MAC* are used to describe network interfaces. In addition, the term *MAC driver* is used to describe network interface drivers.

Although it is not common when discussing VxBus model device drivers, Wind River documentation also uses the term enhanced network driver (or END driver). The term *END driver* refers to a combination that includes both MAC and PHY interfaces. In VxBus, MAC and PHY devices and drivers are handled separately so the term END driver is not generally used.

### 6.1.2 Networking Overview

This section discusses basic networking concepts that are relevant to device driver development. For a more complete discussion of networking in VxWorks and for more information on networking interfaces, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide, Volume 3: Interfaces and Drivers*.

#### Seven Layer OSI Model

Open Systems Interconnection (OSI) is an organization that defines and publishes a model of network software. The published model consists of seven layers, as shown in Figure 6-3. This definition of layers is used throughout Wind River documentation when discussing network stacks and drivers.

Figure 6-1 **Seven Layer OSI Model**

| |
|---|
| Layer 7 – Application |
| Layer 6 – Presentation |
| Layer 5 – Session |
| Layer 4 – Transport (for example, TCP and UDP) |
| Layer 3 – Network |
| Layer 2 – Data Link (MAC and LLC) |
| Layer 1 – Physical |

### Transmission Media and VxWorks

The majority of VxWorks networks use Ethernet as the transmission media. Wind River also supports VxWorks network drivers and configurations for shared memory and for serial network transports. While other network transports are possible, this document focuses primarily on Ethernet as the transport.

Most modern VxWorks Ethernet drivers[1] are split into two parts: a MAC driver and a PHY driver. Together, the MAC driver and the PHY driver manage the data link layer within the OSI model. The MAC sub-layer of the data link layer manages protocol access to the physical network medium. This sub-layer deals with extracting data from the wire to send to the protocol, gaining access to the wire to send protocol data, and certain other aspects regarding the transmission of already packetized protocol data.

The PHY sub-layer deals with frame synchronization, flow control, error checking, and other aspects of manipulating individual bits and bytes during transmission.



**NOTE:** The prevalent model of Ethernet 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.

### Protocols

Within VxWorks, network drivers are written to be largely decoupled from the protocol that is being used. This is done by a layer of software between the protocol and the driver. In VxWorks, this is called the multiplexor (MUX). The MUX sits between the network (OSI layer 3) and the data link layer (OSI layer 2).

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

1. Some devices are only capable of a single mode and do not support software link sensing. For example, NE2000 (and compatible) devices support only 10 Mb/s half-duplex links. MAC drivers for such devices do not require the use of any PHY device or PHY driver.

For example, after receiving a packet, the MAC driver does not directly access any structure within the protocol. Instead, the driver calls a MUX-supplied routine that handles the details of passing the data up to the protocol.

## 6.2 Network Interface Drivers

This section assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class. You should also be familiar with the Wind River Network Stack and its associated documentation.

### 6.2.1 Network Interface Driver Overview

This section presents a basic overview of how network interface drivers function in a VxWorks system.

**NOTE:** Network interface drivers are commonly referred to in this document as MAC drivers.

#### Functional Modules

A MAC driver's basic components include:

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

These basic functions are described further in the following sections.

### Reception

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

- Accept incoming frames from a DMA engine.
- Pass the incoming frames to the MUX.
- Provide the DMA engine with a continuous supply of DMA buffers.

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

Each instance of a MAC driver has a private buffer pool into which incoming DMAs are directed. A MAC driver loans individual buffers from its pool to the stack. There is no guarantee that any individual buffer will be returned to the driver after having been loaned to the network stack.

### Transmission

The driver 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.

A protocol requests that a MAC 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 are complete.

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.

### Command and Control Module

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

The MAC driver command and control module is the part of the driver that parses the driver configuration parameters, quiesces the device, and configures the device in the prescribed mode. It incorporates the driver's load, unload, start, stop, and **ioctl( )** routines, as well as routines for querying and modifying the multicast filter. In essence, the driver's command and control provides the driver's external

interface, with the exception of send and receive. The driver interrupt service routine (ISR is considered a part of the driver command and control module.

#### Network Driver Interrupts

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 packet is transmitted.
- Handling error conditions.

There are two common configurations of interrupts for network devices. Network devices can provide a single interrupt line for all types of interrupts. Or they can provide one interrupt line each for transmit events, receive events, and error events.

When your network device provides only a single interrupt line for all types of interrupts, only a single ISR can be called to service all types of interrupts. When this ISR is called, it 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.

The task-level routines for each type of interrupt should process all the work that is available for that particular type, as discussed in *Receive Handler Interlocking Flag*, p.115 and *Fair Received Packet Handling*, p.116.

### 6.2.2 VxBus Driver Methods for Network Interface Drivers

MAC drivers are required to support the **{muxDevConnect}( )** driver method. This driver type is also likely to use several other methods including: **{vxbDrvUnlink}( )**, **{miiMediaUpdate}( )**, **{miiRead}( )**, **{miiWrite}( )**, and **{miiLinkUpdate}( )**.

The media independent interface (MII) driver methods provide a means of communication between a MAC driver and a PHY driver. The full interface involves driver methods provided by MAC drivers and invoked by PHY drivers (described here) and additional driver methods provided by PHY drivers and invoked by MAC drivers (see *6.3.2 VxBus Driver Methods for PHY Drivers*, p.126).

### {muxDevConnect}( )

The **{muxDevConnect}( )** driver method provides the mechanism for binding the network device to the network stack. This method is invoked on every MAC instance during the network stack initialization. The contents of **func{muxDevConnect}( )** are nearly identical in all MAC drivers.

The following is an example from the **ns83902VxbEnd** driver:



```
/*******************************************************************************
*
* nicMuxConnect - muxConnect method handler
*
* This function handles muxConnect() events, which may be triggered
* manually or (more likely) by the bootstrap code. Most VxBus
* initialization occurs before the MUX has been fully initialized,
* so the usual muxDevLoad()/muxDevStart() sequence must be defered
* until the networking subsystem is ready. This routine will ultimately
* trigger a call to nicEndLoad() to create the END interface instance.
*
* RETURNS: N/A
*
* ERRNO: N/A
*/

LOCAL void nicMuxConnect
    (
    VXB_DEVICE_ID pInst,
    void * unused
    )
    {
    NIC_DRV_CTRL *pDrvCtrl;

    pDrvCtrl = pInst->pDrvCtrl;

    /* Save the cookie. */

    pDrvCtrl->nicMuxDevCookie = muxDevLoad (pInst->unitNumber,
        nicEndLoad, "", TRUE, pInst);

    if (pDrvCtrl->nicMuxDevCookie != NULL)
        muxDevStart (pDrvCtrl->nicMuxDevCookie);

    return;
    }
```

The differences between drivers typically include:

- Changing **NIC_DRV_CTRL** to the **pDrvCtrl** structure definition used by the driver.
- Changing **necEndLoad( )** to the load routine defined in the driver.

- Changing the references of **pDrvCtrl->nicMuxDevCookie** to whatever is appropriate for the driver.

In addition, the **func{muxDevConnect}( )** routine may deal with MIB2 statistics, with creation of the MII bus, or with the presence of multiple links on a single device.

#### {vxbDrvUnlink}( )

The **{vxbDrvUnlink}( )** driver method requests that an instance be shut down. This can occur if your VxBus instance is terminated, or if the driver is unloaded. When an unlink event occurs, you must do the following:

- Shut down and unload the driver interface associated with this device instance.
- Release all the resources allocated during instance creation, such as **vxbDma** memory and maps.
- Shut down and unload all interrupt handles associated with this instance.

If the driver created an MII bus, you must also destroy the child **miiBus** and PHY devices.

#### {miiMediaUpdate}( )

The **{miiMediaUpdate}( )** driver method allows the **miiMonitor** task to notify your driver of link state changes. The **func{miiMediaUpdate}( )** routine is invoked by the **miiMonitor** task when it detects a change in link status. Normally, the **miiMonitor** task checks for link events every two seconds.

Once you have determined the new link state, you must announce the change to any bound protocols using **muxError( )**. You must also update the **ifSpeed** fields in the MIB2 structures, if used, so that any SNMP queries can detect the correct link speed.

#### {miiRead}( )

The **{miiRead}( )** driver method allows PHYs on the MII bus to access your device's MII management registers. The **{miiRead}( )** method is defined as follows:

```
LOCAL STATUS miiRead
    (
    VXB_DEVICE_ID pInst,
    UINT8 phyAddr,
    UINT8 regAddr,
    UINT16 * dataVal
    )
    {
    return (OK);
    }
```



Each MAC driver should provide an MII bus read routine so that the MII bus code can perform management data input/output (MDIO) read accesses to connected PHYs. The **pInst** parameter supplied as an argument to the **{miiRead}( )** method is that of the parent MAC device (which is provided when the parent MAC driver creates a bus with the **miiBusCreate( )** routine). The **phyAddr** argument indicates which PHY address is to be queried, and can be any value from 0 to 31. The **regAddr** argument indicates which register is to be read, and can also be any value from 0 to 31. The **dataVal** argument points to a 16-bit storage location where the **func{miiRead}( )** routine will place the value read from the specified register. If the **func{miiRead}( )** method fails and returns **ERROR**, then **dataVal** is set to 0xFFFF.

It is possible for the **{miiRead}( )** method to return successfully but not return any valid data. For example, if a request is made to read register 1 on the PHY at address 10, but there is no PHY actually available at that address, then the MDIO access may succeed, but no valid register information is obtained. In this case, the hardware typically returns a value of 0xFFFF. The MII bus probe code checks for this case and only assumes that valid data is returned if both the **{miiRead}( )** method succeeds and the register value is not 0xFFFF.

### {miiWrite}( )

The **{miiWrite}( )** driver method allows PHYs on the MII bus to access the device MII management registers. The **{miiWrite}( )** method is defined as follows:

```
LOCAL STATUS miiWrite
    (
    VXB_DEVICE_ID pInst,
    UINT8 phyAddr,
    UINT8 regAddr,
    UINT16 dataVal
    )
    {
    return (OK);
    }
```

The **{miiWrite}( )** method is the complement to the **{miiRead}( )** method, allowing the **miiBus** to perform MDIO write accesses to connected PHYs. The **pInst**,

**phyAddr**, and **regAddr** arguments are the same as they are for **{miiRead}( )**. The **dataVal** argument is the value to be written into the register. The **func{miiWrite}( )** routine should only return **OK** if the register is successfully updated.

#### {miiLinkUpdate}( )

The **{miiLinkUpdate}( )** method is defined as follows:

```
LOCAL STATUS miiLinkUpdate
    (
    VXB_DEVICE_ID pInst
    )
    {
    return (OK);
    }
```

This method is invoked by the MII bus layer whenever **miiBusMonitor** task detects a link state change, which is either a transition from link up to link down, or from link down to link up. The **pInst** argument is a pointer to the MAC driver's instance handle. The MAC driver can use the **{miiLinkUpdate}( )** method to perform any operations that are required when a link state change occurs. This can include setting the MAC speed to match the link speed, enabling or disabling full duplex mode, or configuring flow control. (If the hardware is designed to handle link state changes automatically and does not need any software assistance, these steps can be omitted.) The driver can query the current link state using the **miiBusModeGet( )** routine.

The **{miiLinkUpdate}( )** method is also typically used by drivers to notify bound protocols of link state changes. The VxBus MAC drivers included with VxWorks use the **muxError( )** routine to send either an **END_ERR_LINKUP** or **END_ERR_LINKDOWN** notification to the MUX, which is propagated to all protocols currently bound to the interface.

In addition, MAC drivers can optionally use the **{miiLinkUpdate}( )** method to update MIB information so that the correct interface speed values are reported to SNMP queries.

### 6.2.3 Header Files for Network Interface Drivers

There are several header files typically included for MAC drivers. They fall, loosely, into three categories. The groupings are as follows:

- **Network Stack Interface**

```
#include <end.h>
#include <endLib.h>
#include <endMedia.h>
#include <etherMultiLib.h>
#include <in_cksum.h>
#include <muxLib.h>
#include <netBufLib.h>
#include <net/if.h>
#include <netinet/if_ether.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/in_var.h>
#include <netinet/ip.h>
#include <netLib.h>
#include <net/mbuf.h>
#include <net/protosw.h>
#include <net/route.h>
```

- **MIB2 Interface**

```
#include <m2IfLib.h>
#include <m2Lib.h>
```

- **VxBus Utilities**

```
#include <hwif/util/vxbDmaBufLib.h>
#include <hwif/util/vxbDmaLib.h>
#include <hwif/util/vxbParamSys.h>
```

**NOTE:** Not all MAC drivers are required to include all of the header files listed in this section.

### 6.2.4 BSP Configuration for Network Interface Drivers

There are three standard BSP resources for network devices, all dealing with the interface between MAC drivers and PHY devices. These are: **phyAddr**, **miiIfName**, and **miiIfUnit**.

#### phyAddr

Each Ethernet network device must be connected to one or more PHY devices in order for information to be transmitted out of the hardware. The PHY devices reside on a separate MII bus, and each PHY device has an address associated with it. The connection between the network interface and PHY devices can be hardwired on the target hardware.

The **phyAddr** resource contains the MII bus address of the PHY device (or set of PHY devices) to which the MAC driver is connected. This information is used when communicating with the PHY driver.

Multiple PHY devices can be connected to a single network interface. If only one is connected, the resource is called **phyAddr**. If more than one PHY is connected to a single network interface, the devices are referred to as **phyAddr***N*, where *N* is a small number indicating where in the sequence the device is connected.

For more information, see *6.3 PHY Drivers*, p.123.

#### miiIfName and miiIfUnit

In many cases where the network interface is included in a multifunction chip or on a processor chip, the PHY is a completely separate device on the target hardware. In this situation, the PHY is identified by name so that the network interface can find it when the system is booted. The **miiIfName** and **miiIfUnit** resources are used to identify this device.

The code to gain access to the PHY device is as follows:

```
VXB_DEVICE_ID miiDev;
char * miiIfName;
int miiIfUnit;
...
devResourceGet (pHcf, "miiIfName", HCF_RES_STRING, (void *)&miiIfName);
devResourceGet (pHcf, "miiIfUnit", HCF_RES_INT, (void *)&miiIfUnit);
miiDev = vxbInstByNameFind (miiIfName, miiIfUnit);
```

Once **VXB_DEVICE_ID** is known, the MAC driver can find the PHY read and write routines as follows:

```
fccMiiPhyRead = vxbDevMethodGet (miiDev, DEVMETHOD_CALL(miiRead_desc));
fccMiiPhyWrite = vxbDevMethodGet (miiDev, DEVMETHOD_CALL(miiWrite_desc));
```

### 6.2.5 Available Utility Routines for Network Interface Drivers

There are several libraries that provide utility routines for network drivers including:

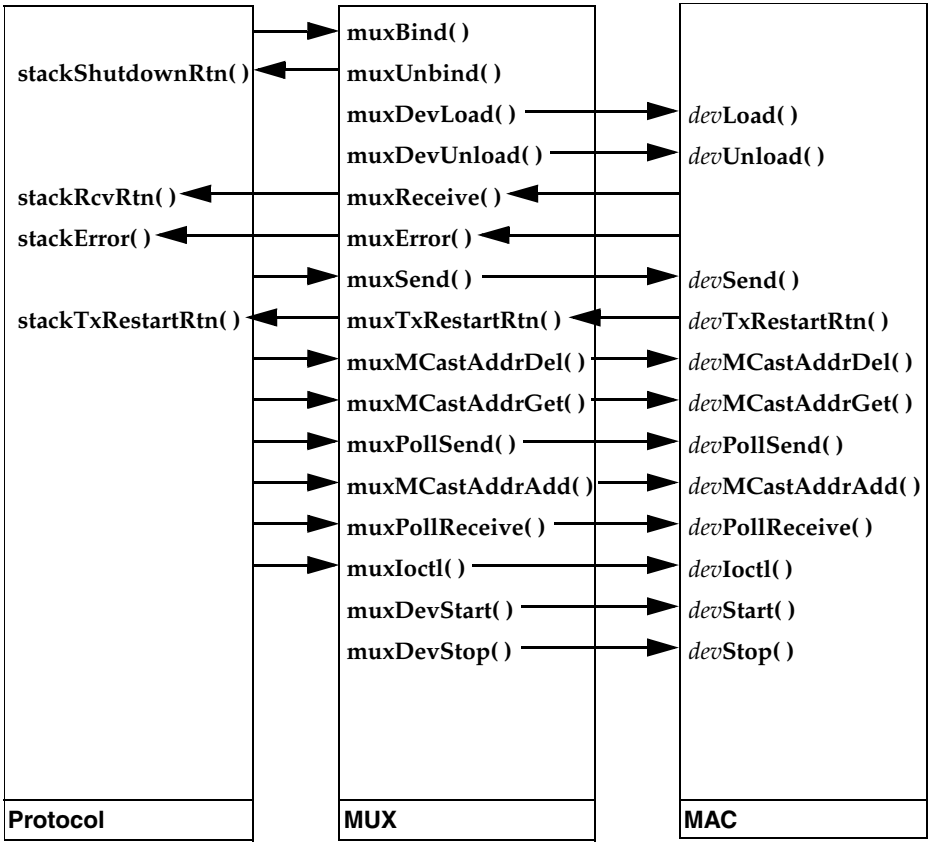
- MUX interactions (**muxLib**)
- job queueing (**jobQueueLib**)
- buffer management (**netBufLib**)
- DMA support (**vxbDmaBufLib**)

These libraries are discussed further in the following sections.

### MUX Interactions

Your driver can use the routines and data structures presented in this section to interact with the MUX. As shown in Figure 6-2, additional MUX routines are available to the network stack. However, those routines are not used by network device drivers.

Figure 6-2 **The MUX Interface**




The routines available to MAC drivers are as follows:

**muxDevLoad( )**
Loads a device into the MUX.

**muxDevStart( )**
Starts a device from the MUX.

**muxIoctl( )**
Accesses control routines.

**muxDevStop( )**
Stops a device.

**muxDevUnload( )**
Unloads a device.

**muxTxRestart( )**
If a device unblocks transmission after having blocked it, this routine calls the **stackTxRestartRtn( )** routine associated with each interested protocol.

### Job Queueing

VxWorks provides the **jobQueueLib** library, which network drivers can use to queue interrupt-driven work to task context. Your driver should do the minimum amount of work in its ISR, and defer most work to task level using **jobQUeueLib**.

**NOTE:** Many desktop and mainframe operating systems use network drivers, which 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). This architecture conflicts with real-time application design. All VxWorks network device drivers must defer packet processing to task context.

The primary **jobQueueLib** routine used by drivers is **jobQueuePost( )**. The prototype for this routine is as follows:

```
STATUS jobQueuePost (JOB_QUEUE_ID jobQueueId, QJOB * pJob)
```

This routine causes the job specified by the **pJob** argument to be executed from the context of a network processing task such as **tNet0**.

**NOTE:** In previous versions of VxWorks, the network processing task was **tNetTask** instead of **tNet0**.

The first argument to **jobQueuePost( )** is a queue ID. Your MAC driver should default to using the default queue ID, **netJobQueueId**. The parameters **rxQueue00** and **txQueue00**, if specified by the BSP or application, are used to find the queue

in place of **netJobQueueId**. The value of the **rxQueue00** and **txQueue00** parameters is a pointer to a structure of type **HEND_RX_QUEUE_PARAM**.

The **jobQueueStdPost( )** routine can be used instead of **jobQueuePost( )** when additional flexibility is required. However, Wind River does not recommend using this routine for performance reasons. For more information, see the reference entry for **jobQueueStdPost( )**.

If there is a requirement for a custom job queue, the **jobQueueCreate( )** and **jobQueueInit( )** routines can be used to create and initialize the custom job queue. However, the standard job queue is sufficient for most network drivers. For more information on **jobQueueCreate( )** and **jobQueueInit( )**, see the corresponding reference entries.



### Buffer Management

The routines in **netBufLib** are used to manage a pool of buffers, along with buffer-specific information contained in structures known as **mBlks** and **clBlks**. The **mBlk** and **clBlk** describe the packet data, and are the structures used by the network stack. The **mBlk**, **clBlk**, and cluster (data buffer) are known collectively as a *tuple*. All data transmitted from the driver to the MUX—for eventual consumption by a network stack—must be held in tuples.

A simplified interface to **netBufLib** is provided by the routines in *installDir*/**vxworks-6.***x*/**target/src/drv/end/endLib.c**. This library provides routines customized to network drivers. These routines are: **endPoolCreate( )**, **endPoolDestroy( )**, **endPoolTupleGet( )**, and **endPoolTupleFree( )**.

The **endPoolCreate( )** utility routine is provided to create a pool suitable for use in a standard network driver. Create the pool as follows:

```
STATUS endPoolCreate
    (
    int tupleCnt,
    NET_POOL_ID * ppNetPool
    )
```

The **tupleCnt** argument specifies the number of tuples required. The **ppNetPool** argument provides a pointer to a storage location to contain the pool ID, for subsequent use by the driver. When jumbo frames are supported, **endPoolJumboCreate( )** can be used in place of **endPoolCreate( )** in order to use 9 KB jumbo clusters instead of the normal 1500 byte clusters.

When the MAC driver is unloaded, it must free the pool resources by calling **endPoolDestroy( )**. The same routine is used whether the pool is configured for standard clusters or jumbo clusters.

```
STATUS endPoolDestroy
    (
    NET_POOL_ID pNetPool
    )
```

Use **endPoolTupleGet( )** and **endPoolTupleFree( )** to allocate and free tuples.

```
M_BLK_ID endPoolTupleGet
    (
    NET_POOL_ID pNetPool
    )

void endPoolTupleFree
    (
    M_BLK_ID pMblk
    )
```

If the simplified interface to **netBufLib** (provided by **endLib**) does not provide sufficient flexibility, the following routines can be used as an alternative:

**netTupleGet( )**
Allocate a tuple from the pool.

**netTupleFree( )**
Return a tuple to the pool.

**netPoolCreate( )**
Create a pool of buffers to hold received packet data.

**netPoolRelease( )**
Release a pool of buffers when unloading the driver.

**netBufLib** also includes routines to allocate individual **mBlks**, **clBlks**, and clusters, which the driver can then link together to form a tuple. However, for performance reasons, Wind River recommends that the driver deal only with tuples where possible. For more information on the additional routines, see the reference entry for **netBufLib**.

### DMA Support

The routines in **vxbDmaBufLib** are used to handle address translation and cache issues required by the network device to support DMA operations. These routines are described in *VxWorks Device Driver Guide (Vol. 1): Device Driver Fundamentals*.

### PHY and MII bus interactions

When MAC drivers initialize the link, part of the required initialization includes determining what PHY device is present and configuring that PHY device. A common procedure is to create a subordinate MII bus, create a list of PHY devices on the MII bus, and determine which devices are present and which device is connected to the network. The following routines are available to help with this MII bus management. For more information, see the reference documentation for **miiBus.c**.



#### miiBusCreate( )

The **miiBusCreate( )** routine is defined as follows:

```
STATUS miiBusCreate
    (
    VXB_DEVICE_ID pInst,
    VXB_DEVICE_ID *pBus
    )
```

This routine creates an MII bus (**miiBus**) instance that is a child of an existing Ethernet device. A pointer to the VxBus instance object representing the device is returned through the **pBus** argument. This bus handle should be saved so that it can be used with other routines. Once the MII bus instance is created, the bus is probed for all PHY devices. When any PHY device is discovered, a VxBus instance is created for it automatically. The **miiBusCreate( )** routine should not be called until the MAC driver's **{miiRead}( )** and **{miiWrite}( )** methods are able to perform MDIO read and write accesses. (That is, the hardware is sufficiently initialized that these accesses succeed.)

This routine is normally called during MAC driver initialization. Once created, the bus should remain until the MAC driver is unloaded.

**NOTE:** The first time **miiBusCreate( )** is invoked, it spawns the **miiBusMonitor** task.

#### miiBusDelete( )

The **miiBusDelete( )** routine is defined as follows:

```
STATUS miiBusDelete
    (
    VXB_DEVICE_ID pInst
    )
```

This routine deletes an **miiBus** object from VxBus, along with all of its child PHY device objects. The **pInst** argument is a pointer to the **miiBus** device instance that

was originally provided to the caller by the **miiBusCreate( )** routine. This routine should be called as part of a MAC driver's unlink process. Once all child devices have been removed, the storage for the **miiBus** is also released.

**NOTE:** When the last **miiBus** in the system is deleted, the **miiBusMonitor** task is also deleted.

### miiBusModeGet( )

The **miiBusModeGet( )** routine is defined as follows:

```
STATUS miiBusModeGet
    (
    VXB_DEVICE_ID pInst,
    UINT32 * pMode,
    UINT32 * pSts
    )
```

This routine is provided as an interface to the **{miiModeGet}( )** methods exported by individual PHY drivers. The **pMode** and **pSts** arguments are specified in exactly the same manner as the arguments to **func{miiModeGet}( )** described in *6.3.2 VxBus Driver Methods for PHY Drivers*, p.126. However, the **pInst** argument in this case is a pointer to the **miiBus** instance context rather than the PHY instance context. This routine is used by Ethernet MAC drivers to query the current link state and characteristics. This in turn leads to calls to the MAC driver's **{miiRead}( )** method to access the PHY.

### miiBusModeSet( )

The **miiBusModeSet( )** routine is defined as follows:

```
STATUS miiBusModeSet
    (
    VXB_DEVICE_ID pInst,
    UINT32 mode
    )
```

Like **miiBusModeGet( )**, this routine provides an interface to the **{miiModeSet}( )** methods exported by individual PHY drivers. This routine is typically called from a MAC driver's start routine to initialize the link to a known state (typically **autoneg**). It can be called to change the link state to any desired settings at any time.

Note that in order to reduce the number of register accesses performed, you should call **miiBusModeGet( )** and **miiBusModeSet( )** as infrequently as possible. For example, your MAC driver could call **miiBusModeGet( )** only when its **{miiLinkUpdate}( )** method is invoked and then cache the results, rather than calling **miiBusModeGet( )** every time its **EIOCGIFMEDIA ioctl( )** handler is called.

#### miiBusMediaListGet( )

The **miiBusMediaListGet( )** routine is defined as follows:

```
STATUS miiBusMediaListGet
    (
    VXB_DEVICE_ID pInst,
    END_MEDIALIST ** ppMediaList
    )
```

This routine returns a pointer to an **END_MEDIALIST** structure that contains entries for all of the media supported by the PHYs on the specified MII bus. A MAC driver can use this information when providing responses to **EIOCGIFMEDIALIST ioctl( )** queries. This structure also includes a default media setting which specifies the default media type for this bus. Typically, the default value is **IFM_AUTO**.



### 6.2.6 Initialization for Network Interface Drivers

In initialization phase 1, network drivers should be sure to disable interrupts for any device that could generate interrupts before an ISR is connected. The remaining network device initialization occurs in phase 2. During phase 2, the kernel is up and kernel features such as standard memory allocation are available.

The final phase of network device initialization is to connect the device to the MUX so that the network stack can gain access to it. This is performed outside the normal VxBus initialization scheme, using the **{muxDevConnect}( )** method (see *{muxDevConnect}( )*, p.89). The actual initialization occurs from the network initialization code. For more information, see *6.2.7 MUX: Connecting to Networking Code*, p.102.

When the network stack initialization code calls the driver's **func{muxDevConnect}( )** routine, the routine calls **muxDevLoad( )**, specifying the **endLoad( )** entry point into the driver. The **muxDevLoad( )** routine calls the specified **endLoad( )** entry point two times. The first time the **endLoad( )** entry is called, the argument contains a pointer to an empty string. The driver must write the driver name into the string for use by the MUX. In the second call, the argument contains a pointer to a non-empty string. At this time, the driver should complete any remaining initialization.

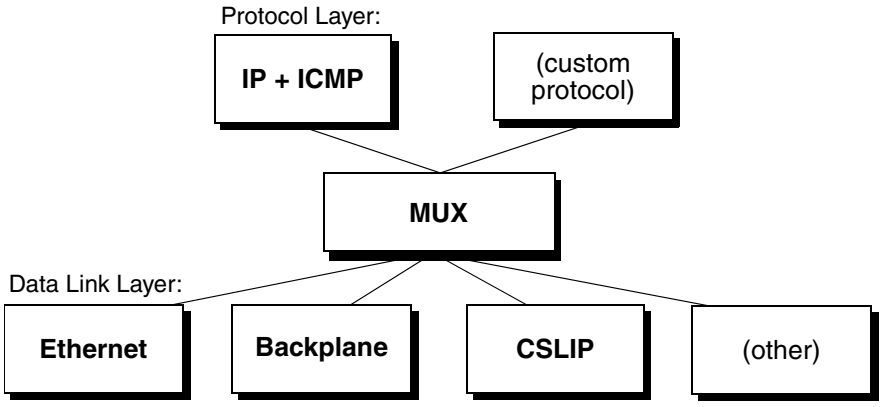
After control returns from **endLoad( )** to **muxDevLoad( )**, the MUX completes the **END_OBJ** structure (by giving it a pointer to a routine your driver can use to pass packets up to the MUX). The MUX then adds the returned **END_OBJ** to a linked list of **END_OBJ** structures. This list maintains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

devices on the system. After control returns from **muxDevLoad( )**, your driver is loaded and ready to use.

### 6.2.7 MUX: Connecting to Networking Code

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. A MAC driver does not directly interface with the data link layer, but rather interfaces with the MUX, which is an abstraction layer that de-couples the driver from any particular protocol. This API multiplexes access to the networking hardware for multiple network protocols. Figure 6-3 shows the MUX in relationship to the protocol and data link layers.

Figure 6-3 **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 driver initialization is done outside of the normal VxBus initialization phases. This is because the driver connects to the MUX. The MUX is initialized during network configuration, and network drivers must be connected to the MUX during network configuration in order for a network device to be used as a boot device. Therefore, the network driver must not attempt to connect to the MUX

until network initialization has initialized the MUX. Driver initialization must be completed before network initialization is complete.

The solution to this is the use of the **{muxDevConnect}( )** driver method. This method is called for all network devices from the network initialization code, after the MUX is initialized. The legacy network initialization using **endDevTbl[ ]** is performed after VxBus network drivers have initialized.



### 6.2.8 jobQueueLib: Deferring ISRs

When working with VxWorks network drivers, you must understand why network drivers defer their ISR processing to task context, and how they accomplish this.

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. This is accomplished with the use of **jobQueueLib**.

#### Interrupt Handlers

Upon arrival of an interrupt on a network device, VxWorks invokes the driver's previously registered ISR. Your driver ISR should do the minimum amount of work necessary to get the packet off of the local hardware. To minimize interrupt lock-out time, the ISR should handle only those tasks that require minimal execution time, such as error or status change. The 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 **jobQueuePost( )**. (For information on **jobQueuePost( )**, see *Job Queueing*, p.96.)

**NOTE:** You can also use **jobQueuePost( )** to queue up work other than processing of received packets.

**CAUTION:** Use **jobQueuePost( )** sparingly. The ring buffer used to queue jobs is a finite resource that is also used by the network stack. If it overflows, the network stack may be corrupted.

### 6.2.9 netBufLib: Transferring Data with mBlks

Network drivers pass received packet data to the MUX and receive packet data to transmit from the MUX. The data are kept in structures called **mBlks**. The routines in **netBufLib** provide a means of managing the **mBlks** structures and the data they contain.

Each network interface requires its own memory pool for data and **mBlks**. Received data is put in the data blocks allocated from this pool, and sent to the MUX. Data for transmission is allocated from the system pool. Once the data are sent, the driver must free data blocks and **mBlks**, so they can be returned to the system pool.

The term *cluster* is used to refer to buffers containing packet data.

#### Setting Up a Memory Pool

Each MAC driver unit requires its own memory pool. Network interface drivers typically use a pool with a single fixed block size, so that a single cluster is large enough to hold an entire received packet with little or no wasted space. The exception is when scatter-gather is supported, in which a smaller cluster size can be used. (For more information on scatter-gather, see *Supporting Scatter-Gather*, p.105.) An Ethernet network's MTU is typically 1500 bytes unless jumbo frames are supported and configured, and a typical network driver cluster size is 1500 or 1540 bytes. (For more information on memory pools, see the reference entry for **netBufLib**.)

The following code is a simplified version of the code in the **mvYukonIIVxbEnd** driver used to create a driver pool. This code checks whether the driver should be configured to use jumbo frames, and allocates a pool based on this information.

```
stat = vxbInstParamByNameGet(pDev, "jumboEnable",
                                    VXB_PARAM_INT32, &jumboSupported);
```

```
if (stat != OK || jumboSupported == 0)
    {
    pDrvCtrl->ynMaxMtu = YN_MTU;
    stat = endPoolCreate (768, &pDrvCtrl->ynEndObj.pNetPool);
    }
else
    {
    pDrvCtrl->ynMaxMtu = YN_JUMBO_MTU;
    stat = endPoolJumboCreate (768, &pDrvCtrl->ynEndObj.pNetPool);
    }

/* Allocate a buffer pool */

if (stat == ERROR)
    {
    logMsg("%s%d: pool creation failed\n", (int)YN_NAME,
            pDev->unitNumber, 0, 0, 0, 0);
    return (NULL);
    }
```

Regardless of whether or not jumbo frames are enabled, this driver specifies the constant value of 768 tuples. A further enhancement would be to make this a configurable parameter.

Once the pool is created, the driver can allocate tuples with a call to **endPoolTupleGet( )**.

```
pMblk = endPoolTupleGet (pDrvCtrl->ynEndObj.pNetPool);
```

### Supporting Scatter-Gather

Scatter-gather is a DMA technique that allows for a single large block of data to be distributed among multiple clusters. When data is being sent, the DMA engine within the device gathers data from each cluster in turn, and sends it out to the output stream. When data is being received, the DMA engine within the device scatters data into multiple clusters, filling each cluster in turn. For performance reasons, if a device supports scatter-gather, the driver for the device should support scatter-gather as well, at least for output.

When transmitting packets, the network stack is often unable to find a single cluster that is large enough to hold a large packet. When this happens, it obtains multiple tuples with clusters that have sufficient total space to hold the packet and links them together to form an **mBlk** chain. The network stack copies the data into the chained clusters, then sends the fragmented packet to the driver as an **mBlk** chain.

When scatter-gather is not supported and the network stack sends a fragmented packet to the driver, the driver must coalesce the **mBlk** chain. This involves

allocating a cluster from the driver pool and copying the packet fragments from the **mBlk** chain into the newly allocated cluster. Because VxWorks network drivers typically create pools with clusters large enough to hold an entire packet, the fragmented data fits within a single cluster. However, this copy operation is time consuming and results in lower network throughput.

When a device supports scatter-gather for transmit, 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. This is typically faster than the copy operation required when the **mBlk** chain is coalesced.

Each fragment, or link of the **mBlk** chain, requires a DMA descriptor or a portion of a DMA descriptor, depending on the design of the device. If there are not enough free descriptors to hold the **mBlk** chain, or if there are not enough scatter-gather slots in the descriptor, then the **mBlk** chain cannot be sent using scatter-gather, but must be coalesced into a single cluster. The driver send routine is responsible for determining whether the **mBlk** chain can be sent using scatter-gather, or whether it needs to be coalesced.

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 number of available descriptors or the number of scatter-gather slots available in a descriptor. If the **mBlk** chain cannot be sent using scatter-gather, the driver must either coalesce the **mBlk** chain into a single cluster allocated from the driver's pool, or it must discard the packet.

In most VxWorks network 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, VxWorks network drivers do not fragment incoming frames. However, if the device supports segmenting incoming frames across multiple clusters, and if the driver's memory consumption is more important than throughput, the driver can be written to take advantage of this feature thus reducing the pool memory requirement at the risk of reducing throughput.

### 6.2.10 vxbDmaBufLib: Managing DMA

Most network devices provide built-in DMA engines, which the driver must manage. There are several issues related to supporting these DMA engines, primarily related to address translation of data buffers and of CPU cache configuration.

VxBus provides support for drivers to manage both of these considerations in the **vxbDmaBufLib** library. For more information on this library, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.

### 6.2.11 Protocol Impact on Drivers

Although the MUX enforces a distinction between the driver and the protocol, there are nevertheless a few restrictions that the protocol imposes on drivers and a few protocol-specific optimizations available to drivers for use where appropriate. This section lists these restrictions and optimizations.



#### IPv4 and IPv6 Checksum Offloading

TCP/IP checksum offloading eliminates host-side check summing overheads by performing 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**
Eight 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, therefore 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 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 ioctl( )** routines in the driver.

The driver configures the device 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.

```
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 **ioctl( )** to determine what capabilities the driver supports. The network stack can 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 **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:

```
(CSUM_IP|CSUM_TCP|CSUM_UDP)
```

Interface capability flags for the **cap_available** and **cap_enabled** fields are listed in Table 6-1.

Table 6-1 **Interface Capability Flags for cap_available and cap_enabled**

| Flag | Description |
|---|---|
| **IFCAP_RXCSUM** | Supports IPv4 receive checksum offload |
| **IFCAP_TXCSUM** | Supports IPv4 transmit checksum offload |
| **IFCAP_NETCONS** | Supports being a network console |
| **IFCAP_VLAN_MTU** | Supports VLAN-compatible MTU |
| **IFCAP_VLAN_HWTAGGING** | Supports hardware VLAN tags |
| **IFCAP_JUMBO_MTU** | Supports 9000 byte MTU |
| **IFCAP_TCPSEG** | Supports IPv4/TCP segmentation |
| **IFCAP_IPSEC** | Supports IPsec |
| **IFCAP_RXCSUMv6** | Supports IPv6 receive checksum offload |

Table 6-1 **Interface Capability Flags for cap_available and cap_enabled** (cont'd)

| Flag | Description |
|---|---|
| **IFCAP_TXCSUMv6** | Supports IPv6 transmit checksum offload |
| **IFCAP_TCPSEGv6** | Supports IPv6/TCP segmentation |
| **IFCAP_IPCOMP** | Supports IPcomp |
| **IFCAP_CAP0** | Vendor specific capability #0 |
| **IFCAP_CAP1** | Vendor specific capability #1 |
| **IFCAP_CAP2** | Vendor specific capability #2 |

Flags indicating hardware checksum support and software checksum requirements are listed in Table 6-2.

Table 6-2 **Hardware and Software Checksum Support Flags**

| Flag | Description |
|---|---|
| **CSUM_IP** | Enable IP checksum |
| **CSUM_TCP** | Enable TCP checksum |
| **CSUM_UDP** | Enable UDP checksum |
| **CSUM_IP_FRAGS** | Enable checksum of IP fragments (currently not supported) |
| **CSUM_FRAGMENT** | Stack can fragment IP packets |
| **CSUM_TCP_SEG** | Stack can segment TCP/IPv4 |
| **CSUM_TCPv6** | Enable checksum for TCP/IPv6 |
| **CSUM_UDPv6** | Enable checksum for UDP/IPv6 |
| **CSUM_TCPv6_SEG** | Stack can segment TCP/IPv6 |
| **CSUM_IP_CHECKED** | IP checksum done |
| **CSUM_IP_VALID** | IP checksum is valid |
| **CSUM_DATA_VALID** | **csum_data** field is valid |

Table 6-2 **Hardware and Software Checksum Support Flags** (cont'd)

| Flag | Description |
|---|---|
| **CSUM_PSEUDO_HDR** | **csum_data** has pseudo header |
| **CSUM_DELAY_DATA** | **(CSUM_TCP \| CSUM_UDP)** |
| **CSUM_DELAY_IP** | **(CSUM_IP)** |
| **CSUM_DELAY_DATA6** | **(CSUM_TCPv6 \| CSUM_UDPv6)** |
| **CSUM_RESULTS** | **(CSUM_IP_CHECKED \| CSUM_IP_VALID \| CSUM_DATA_VALID \| CSUM_PSEUDO_HDR)** |
| **CSUM_IP_HDRLEN** | **(pMblk) ((pMblk)->mBlkHdr.offset1)** Used to determine the actual IP header length |
| **CSUM_XPORT_HDRLEN** | **(pMblk) (((pMblk)->mBlkPktHdr.csum_data & 0xff00) >> 8)** Location of TCP or UDP checksum field |
| **CSUM_XPORT_CSUM_OFF** | **(pMblk) ((pMblk)->mBlkPktHdr.csum_data) &** |

The **END_CAPABILITIES** structure is used by the driver's interface capabilities set and get **ioctl( )** routines:

**EIOCSIFCAP**
Interface capabilities set **ioctl( )**.

**EIOCGIFCAP**
Interface capabilities get **ioctl( )**.

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

**EIOCGIFCAP** example:

```
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 in **cap_enabled**. This allows the stack to turn capabilities on or off as required. The stack can request any capability that it is capable of supporting. 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:

```
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 does the following:

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 and does the following:
   - Determines if the device calculated the IP checksum.

     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.
   - Tests to see if the device determined that the IP header is valid.

     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.

- 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.
  - 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 has been calculated and that the packet is valid.
  - 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 has been computed.
  - If the driver determines that the packet and checksum are valid, it writes the checksum into the **mBlk** at **pMblk->m_pkthdr.csum_data**. The driver does not need to read the calculated checksum from a device register. A valid checksum value is 0xffff, the driver can write this value into the **mBlk**.



### Handling Corrupt Packets

If a packet is corrupted, the driver has two options. It can chose 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:

```
/* 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_CSUMSTS_OFFSET, csumStatus);

    /* Determine if IP checksum calculated */

    if (csumStatus & RFD_CS_IP_CHECKSUM_BIT_VALID)
        {
         /* Set mBlk check sum 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 check sum 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**.
   - If the network stack requests that the device calculate the IP checksum, the driver prepares to set the device accordingly.
   - 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:

```
/* 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_DELAY_DATA)
        txCsum |= IPCB_TCPUDP_CHECKSUM_ENABLE;
    if (pMblkHead->m_pkthdr.csum_flags & CSUM_TCP)
        txCsum |= IPCB_TCP_PACKET;
    }
CFD_WORD_WR (pCFD, CFD_IPSCHED_OFFSET, txCsum);
}
```



### 6.2.12 Other Network Interface Driver Issues

This section discusses additional issues that must be handled by your driver.

#### Receive Handler Interlocking Flag

VxWorks network drivers defer much of the work related to servicing interrupt conditions to code executing in a task level context by calling **jobQueuePost( )**.

Because interrupts are relatively costly in terms of overall system performance, one goal of a network interface driver is to minimize the number of interrupts that occur. This can be handled, in part, in the task-level service routine executed from **jobQueuePost( )**. This routine should continue to service pending events for as long as work is available. The ISR configures the device so that it does not generate interrupts and then makes the call to **jobQueuePost( )** to schedule the task-level service routine. The task-level service routine continues running as long as there are incoming packets to dispatch to the stack.

The ring buffer used to implement the **jobQueuePost( )** facility is a limited resource therefore, when writing your device driver, you must take great care to safeguard against overflowing the ring buffer. If the ring is allowed to overflow, the network stack can be corrupted. When the network device does not share its interrupt line with any other device, there are no problems safeguarding the ring buffer, because the device is configured not to generate interrupts. However, when the interrupt line is shared with some other device, the network driver's ISR can be called even when the device has not generated an interrupt. Unless something is done to prevent the call, the ISR calls **jobQueuePost( )** a second time, even though one copy of the task-level service routine is already scheduled or running. If many of these events occur in the system, the ring buffer overflows, and stack corruption occurs.

The solution to this problem is to provide a receive handler interlock flag. This is a simple boolean flag, kept in the instance's **pDrvCtrl** structure. This flag indicates whether the task-level service routine is already scheduled. If the interlock flag is set, the ISR does not make a second, superfluous call to **jobQueuePost( )**.

The order of execution for use of the receive handler interlock flag is as follows:

1. Within the ISR, the code takes a spinlock.
2. The code checks the interlock flag.
   a. If the receive handler interlock flag is set, the ISR releases the spinlock and returns.
   b. If the receive handler interlock flag is not already set, the ISR sets the flag, releases the spinlock, and schedules the task-level service routine with a call to **jobQueuePost( )**.
3. Within the task-level service routine, the code processes received packets[2].
4. When all received packets are processed[3], the task-level service routine takes the spinlock.
5. The task-level service routine checks whether there are additional incoming packets that have been received since the previous check.
   a. If there are additional incoming packets, the task-level service routine releases the spinlock and reschedules itself with a call to **jobQueuePost( )**.
   b. If there are no additional incoming packets, the task-level service routine clears the receive handler interlock flag and releases the spinlock.

#### Fair Received Packet Handling

In the basic implementation, a task-level service routine processes all incoming packets before exiting. However, during periods of peak traffic on high bandwidth networks, this can result in a performance bottleneck that degrades the overall system performance. Because the driver sends packets until no additional packets are available, it can take quite a while for the entire descriptor queue to be processed. During that time, the network stack and all other drivers may not be

2. The driver can choose to process received packets until there are no more to be done, or it can choose to implement the fair received packet handling scheme (see *Fair Received Packet Handling*, p.116).
3. If fair received packet handling is implemented, received packets may not be processed until after several calls to **jobQueuePost( )** are made.

able to perform any work. For this reason, you should limit the number of received packets to a fixed maximum that can be specified with a driver parameter.

To implement this scheme, the driver's task-level service routine must process incoming packets until either there are no additional incoming packets, or until the maximum number is reached. Upon completion of this loop, if there are additional received packets, the task-level service routine reschedules itself by calling **jobQueuePost( )** to complete the remaining packets. During the time between the original call to **jobQueuePost( )** and the subsequent call, the network stack and other drivers can queue their own work, which will have a chance to execute before the current driver finishes processing all of its input packets. This generally results in better system performance.



You should also avoid calling **taskDelay( )**—or any other delay mechanism—from the driver task-level service routine. Such a delay prevents processing of packets from other interfaces. For this reason, you must carefully consider the use of delays in your driver. Instead, consider rescheduling the job with another **jobQueuePost( )** 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.

#### Receive Stall Handling

When a device is receiving data packets, it copies the data into buffers (clusters) provided by the driver. Because of the design of the descriptors, the device usually keeps copying data when it arrives, even if some data are being processed at the same time. However, it is possible to receive enough packets to fill the available clusters before the driver has a chance to process the received packets and make additional clusters available. When this happens, the device stops copying into the receive clusters. This condition is known as a receive stall.

When this occurs, devices typically behave in one of two ways:

- Some devices require that the next descriptor in the sequence be cleared, and no additional driver intervention is required. 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 other action on the part of the driver.
- Other devices place their receiver into a halted state by setting a bit in a control register. This type of device often requires the driver to clear the control register bit in addition to freeing the next descriptor, before operation resumes.

## 6.2.13 Debugging Network Interface Drivers

The normal debugging strategies discussed in *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies* apply to network interface drivers. However, there are a number of additional debugging strategies and methodologies available.

In general, when working with a VxWorks network interface driver, you must have some way to boot the VxWorks image without using the driver you are attempting to debug. You can do this by using some other network device on the target hardware, or you can use an image programmed into ROM by some form of hardware debugger.

**NOTE:** The debugging methods described in this section require that you have some means of booting the VxWorks image without using the driver you are debugging.

### Using VxBus Show Routines

Network driver debugging makes more use of the VxBus show routines than any other driver classes. These routines are used in the usual way to find whether the driver matches the device, to find whether the device exists, and so on. However, because PHY device configuration is sometimes performed as part of the network device initialization, PHY debugging is also relevant to network device initialization and VxBus show routines can also be used to help some kinds of PHY debugging.

For example, use **vxBusShow( )** to find whether the appropriate PHY device is connected. To do this, run **vxBusShow(1)** to show the **pRegBase[ ]** entries for each device. The PHY instance entry for **pRegBase[0]** contains the PHY ID of the PHY device.

### Deferring Driver Registration

As with all VxBus drivers, it is helpful to defer driver registration when debugging network drivers. However, in order for the driver to work, you must connect the instance to the MUX and the network stack. In a normal system, these actions are done automatically, once, during system startup, and never done again. Therefore, during testing, you must perform these actions manually.

The easiest way to accomplish this is to use a **vxWorks.st** image, or a project-built image with networking included but disabled. When you are ready to test your driver, call **usrNetInit( )** to perform network initialization, including initialization of your driver.

**NOTE:** When you initialize your driver using **usrNetInit( )** as described above, you do *not* need to connect to the MUX or to the network stack, as described in the following sections.



#### Attaching to the Mux

After registering your driver with VxBus, you need to connect the instance(s) it forms to the MUX. During normal system initialization, this is done automatically from within **usrNetInit( )**. However, if the network is already initialized, this does not work. Instead, you need to call your driver's **func{muxDevConnect}( )** routine manually, passing the instance ID as a parameter. This allows access to your driver from protocols included in the system, but does not attach the protocols.

#### Attaching to the IPv4 Stack

To attach to the IPv4 stack, use a sequence of calls to **ipcom_drv_eth_init( )** and **ifconfig( )** to configure the device and attach to the stack. The routine **ipcom_drv_eth_init( )** uses three arguments: the driver name, the unit number, and a third argument that can always be zero. The routine **ifconfig( )** is similar to **ifconfig( )** on UNIX and similar computers. However, you must include the argument list in quotes. (For a more detailed discussion of the arguments to **ifconfig( )**, see the related reference entry.)

For example, if connecting **YN0** to IPv4 at address 10.0.0.1, use the following commands from the VxWorks development shell:

```
-> ipcom_drv_eth_init("yn", 0, 0)
-> ifconfig("yn0 10.0.0.1 netmask 255.255.255.0 up")
```

### Pairing with a PHY instance

For PHY devices, the **pRegBase[ ]** entries contain the MII addresses of the PHY device. Use **vxBusShow(1)** to show the **pRegBase[ ]** entries and verify that the appropriate PHY device is connected to the MAC instance.

### Stress Testing

Wind River strongly recommends the use of hardware debug tools in order to create reliable network drivers. Using a debug tool such as SmartBits or IXIA, generally allows you to create a better high stress environment for testing and generally leads to a more reliable and robust driver. In many cases, it is not possible to create an adequate test environment without the use of hardware debug tools.

### Netperf Test Suite

In addition to the use of hardware debug tools, a software test platform is can also prove valuable. One such platform, used widely in the industry, is netperf. For information about netperf, and to download the test software, visit the following URL:

**http://www.netperf.org/netperf/**.

### Interrupt Validation

During early parts of debugging, you should instrument the driver's ISR by adding an output message using **logMsg( )**. This lets you know if the device generates interrupts correctly and if the ISR is connected correctly. It also lets you know what types of interrupts are occurring and how the interrupts are being processed.

### Additional Tests

Once your driver provides basic functionality, there are a number of additional tests that can be run. Many of these tests can be used without special hardware or software platforms.

#### Ping-of-Death

Ping-of-death is an attack on drivers based on the value of an unsigned 16-bit field in the ping packet. When the ping packet size is larger than 32 KB (32768 bytes), some drivers and network stacks cannot handle the packet. To generate a ping-of-death, specify a 64 KB ping packet using any ping client.

### Driver Start and Stop

Test starting and stopping your driver. To stop the driver, you can use **ifconfig( )** with the **down** argument and **muxDevStop( )**. The argument to **muxDevStop( )** is the cookie returned by the **muxDevLoad( )** call when you initialize your driver. For example, if your driver's cookie is 0x97830, you can stop the driver as follows:

```
-> ynCookie = 0x97830
-> ifconfig("yn0 down")
-> muxDevStop(ynCookie)
```

To restart the driver, call **muxDevStart( )** and call **ifconfig( )** with the **up** argument:

```
-> muxDevStart(ynCookie)
-> ifconfig("yn0 up")
```

### Driver Load and Unload

Test unloading and reloading your driver from the protocol and MUX. To unload the driver, stop the driver as specified in *Driver Start and Stop*, p.121, then call **muxDevUnload( )**. The routine **muxDevUnload( )** requires the driver name and unit number as arguments. For example:

```
-> ifconfig("yn0 down")
-> muxDevStop(ynCookie)
-> muxDevUnload("yn", 0)
```

To reload the driver, call **muxDevLoad( )** from your driver's **func{muxDevConnect}( )** routine.

You should also test your driver's **func{vxbDrvUnlink}( )** routine if you provided one. Testing this routine can be simplified if you can have the network available using a different driver because this situation allows you to debug your driver without rebuilding the entire VxWorks image.

To test **func{vxbDrvUnlink}( )**, build a VxWorks image without your driver and boot it. Then, download your driver using the **ld** command from the VxWorks development shell. Register your driver with VxBus and connect to the MUX as described in *Deferring Driver Registration*, p.118, then do your testing.

To test any modifications you make during your testing, unload your driver from the MUX, then unload the driver's object module. Next, build a new version of the driver and load the new driver's object module with the **ld** command and perform any additional debugging.

### Polled Mode

You should test your driver's polled mode operation. Typically, the only use of polled mode is by WDB and for core dumps. To test WDB polled mode operation, you can use the WTX test described in *WTX Test*, p.122 to perform some testing.

You should also perform more basic testing before running the WTX test. To do this, write a simple application that makes the **ioctl( )** call to put the instance into polled mode operation. The application should read from the interface using polled mode, modify the packet data, and send the modified packet. This can be testing using ping from your development host.

When polled mode is used to save core dump data, the network is put under heavy load. For this reason, you should run the basic testing with heavy traffic. For example, run ping with large packet sizes from multiple hosts, to ensure that many large packets can be received and transmitted. Additional testing can be done by performing an actual core dump save. For more information on core dump, see the *VxWorks Kernel Programmer's Guide: Error Detection and Reporting*.

### Receive Error Path

Be sure to test the receive error path for your driver. This testing is often overlooked when hardware debug tools are not available, because it is difficult to generate receive error conditions without those tools. You cannot fully validate the receive error path without the assistance of hardware tools.

### WTX Test

You can run a test related to polled mode using WTX. The **WTXTest** uses the target server to connect to the target and performs various stress tests on the driver's polled mode operation. For more information on using WTX test, see *Wind River Workbench User's Guide: Troubleshooting*.

### Multicast Filter Test

If your driver supports multicast, test to see whether the multicast filter works correctly. To do this, you need to write a simple test application that runs on VxWorks, which configures the interface with one or more multicast addresses and sends and receives multicast traffic. You may need host software or multiple VxWorks targets in order to perform this type of test.

# 6.3 PHY Drivers

All 10/100/1000 Ethernet interfaces incorporate a physical layer of some type, commonly known as a PHY. The PHY component may be integrated directly with the MAC, or it may be a separate device connected to the MAC using one of several MAC/PHY media connection types (such as MII, GMII, RGMII, TBI, and so forth). Software interaction with the PHY is necessary in order to properly implement link autoconfiguration and link state change notification within VxWorks.



The MII specification for PHY devices defines a management interface with a total of 32 registers. The first 16 are defined by the specification itself and are common to all devices that comply with the specification. The latter 16 registers are vendor-defined, and vary from one implementation to another. While it is possible to use only the standardized registers to control most devices, there are many cases where use of the vendor-specific API is required. In those cases, it is necessary to implement device-specific PHY software.

Traditionally, both vendor-specific PHY management and link management in general have been implemented largely in an ad-hoc manner. The MII bus and PHY driver mechanism attempts to address this issue by providing both a simple way for Ethernet drivers to handle link management, and for different PHY chips to be handled with discrete, reusable drivers.

## 6.3.1 PHY Driver Overview

The MII bus and PHY layer includes the **miiBus.o** module (which is configured into the system using the **INCLUDE_MII_BUS** component) and various PHY drivers. A given image configuration need only include those PHY drivers that are required to handle the PHY hardware actually present in the system. In many cases, only the **genericPhy** driver is necessary.

The link management functions of MII bus are carried out in the context of the **miiBusMonitor** task. This task periodically checks the state of every interface configured into the system and issues a callback to the corresponding VxBus MAC instance whenever the link state changes.

PHY interrupts are not currently supported. To understand why, consider that acknowledging and cancelling a PHY interrupt requires reading or writing to a PHY register, and that to correctly follow VxWorks driver guidelines, this operation must be done in an ISR. However, PHY register accesses are typically done indirectly through an MDIO port and are not atomic. This means that they must use mutual exclusion protection to prevent overlapping accesses. The

problem with this is that the only mutual exclusion mechanism that works in both task context and interrupt context is **intLock( )** and **intCpuLock( )** (or a spinlock, in VxWorks SMP product), but in some cases a PHY register access can be very slow, and keeping interrupts blocked for the entire period can negatively impact system behavior. (This is especially true where MDIO accesses are performed using *serial bitbang* I/O in software.)

Consequently, PHY register accesses are never done in interrupt context, and polling is used to monitor link state changes instead. The current polling period is two seconds, and **miiBusMonitor** tasks runs at priority 254, in order to reduce load on the system as much as possible.

### PHY Device Probing and Discovery

In a typical system, each MAC instance creates an MII bus, and the MII bus in turn creates one or more PHY instances. The PHY instances are auto-discovered by probing the MII bus. The MII specification allows for up to 32 devices to be uniquely addressed using the MDIO interface. Probing is done by performing a read request of the basic mode status register (register 1) at each of the 32 MII addresses. If reading the register yields a value other than 0 or 0xFFFF, the probe code considers a PHY device to have been found. The probe then creates and announces a VxBus node corresponding to this device. At the time the device instance is created, the PHY ID registers are also read and saved.

Once a PHY instance is announced, VxBus attempts to match a driver to it. This process occurs in two steps. First, VxBus calls the MII bus **miiBusDevMatch( )** routine, which decides whether or not to accept the instance and driver as valid for an MII bus. The **miiBusDevMatch( )** routine almost always returns success as long as the instance declares its bus to be of type **VXB_BUSID_MII**. However, there is one special case. A **genericPhy** driver is available that should work for most MII compliant PHYs. However, there may be a case where both generic PHY and another PHY driver are both registered. The desired behavior is for the **genericPhy** driver to be selected only if no specific driver match is found. However, the **genericPhy** probe routine always returns success. To prevent it from claiming PHY instances unexpectedly, **miiBusDevMatch( )** checks to see if a driver that specifically handles the PHY device is also registered with VxBus. If it finds such a driver, it prevents **genericPhy** from claiming the device so that the other driver can claim it instead.

Once **miiBusDevMatch( )** is called, VxBus invokes the PHY driver probe routine. The probe routine then tests the PHY vendor and device ID against a list of values

supported by the driver. If the driver chooses to claim the device, the probe routine returns **TRUE**.

### MAC and MII Bus Relationship

Each Ethernet MAC driver typically has one MII bus, which the Ethernet driver itself creates using the **miiBusCreate( )** routine. This bus in turn has one or possibly more child PHY devices attached to it. (The simplest and most common case is one Ethernet device instance, with one child MII bus, with one child **genericPhy** instance.)



Originally, the reason for supporting multiple PHYs on a single MII bus was to allow for the design of network controllers with more than one media type. For example, a dual media copper and fiber Ethernet adapter could be built using one Ethernet MAC with two different PHYs (one copper and one fiber). Driver software could set the interface for copper mode by using the MII management interface to isolate or power down the fiber PHY while activating the copper one. Switching to fiber mode could be done using the opposite procedure (isolating the copper PHY and then re-enabling the fiber one). In this configuration, only one PHY is active at a time, and the idle one must be isolated from the MAC data pins.

This configuration is not commonly used (several vendors now support both copper and fiber media in a single PHY chip instead, using vendor-specific programming to switch modes). However, what is more common is the use of a single MDIO port for controlling multiple PHYs connected to different MACs. For example, the Freescale MPC8560 has two built-in TSEC gigabit Ethernet MACs. However, only one of them has a functional MDIO port. This means that software can only access the management registers on the two PHY chips by using the MDIO registers on only one of the TSECs (typically **TSEC0**).

This configuration presents a problem, because it can result in the MII bus for one TSEC device having two child PHY instances, while the MII bus on the other TSEC has none. The Ethernet MAC driver software must be carefully written in order to deal with this condition.

### Generic PHY Driver Support

A special driver, called **genericPhy**, is included with the MII bus subsystem, which is designed to support most 10/100/1000 Mb/s copper PHYs. The **genericPhy** driver uses only those registers defined in the MII specification for controlling the underlying PHY device, and should work with the majority of PHY chips without

modification. The **genericPhy** driver probe routine always succeeds, and acts as a "catch-all" for any PHYs discovered on an MII bus that are not specifically claimed by another driver registered with VxBus.

The **genericPhy** driver always assumes that the PHY device supports at least 10 Mb/s and 100 Mb/s modes in full and half duplex. It also tests for the extended capabilities bit in the status register and, if this bit is set, it enables support for autonegotiation as well.

The **genericPhy** driver is included using the **INCLUDE_GENERICPHY** component.

#### Generic TBI Driver Support

Many fiber optic controllers use a ten bit interface (TBI) as their MAC/PHY media connection. The TBI management interface is similar to that of an ordinary 10/100 copper PHY. However, a TBI PHY supports only 1000 Mb/s. A MAC driver can be written such that the routines in the MII bus library (see the reference documentation for **miiBus**) can discover the TBI management interface and manage the link just like that of an ordinary PHY. Most devices that implement TBI use the same management interface, therefore a **genericTbiPhy** driver is also provided to handle these cases.

Unlike the **genericPhy** driver, the **genericTbiPhy** driver only attaches to a MAC driver that reports the correct vendor and device ID values. The **genericTbiPhy.h** header defines two values, **TBI_ID1** and **TBI_ID2**. If a MAC driver's **{miiBusRead}( )** method returns these values when a caller reads the ID registers, the **genericTbiPhy** driver successfully attaches to the TBI PHY instance.

The **genericTbiPhy** driver is included using the **INCLUDE_GENERICTBIPHY** component.

### 6.3.2 VxBus Driver Methods for PHY Drivers

The MII bus layer has two sets of VxBus methods: upper edge methods, which must be provided by Ethernet MAC drivers, and lower edge methods, which must be provided by child PHY devices that reside on an MII bus.

**NOTE:** These methods typically perform operations that are not atomic. In particular, performing MDIO reads and writes usually requires multiple accesses to MAC registers. Consequently, it is important that these routines internally provide some form of mutual exclusion in order to prevent overlapping accesses. Most VxBus Ethernet drivers do this using a mutex semaphore.



### Upper Edge Methods

Upper edge MII bus methods are typically all exported by the MAC driver that instantiates the MII bus. In most cases, access to the PHY management registers is provided through an MDIO port that is part of the Ethernet MAC itself. This can either be a low level bitbang interface to the MDIO pins, or it can be a set of "shortcut" registers that permit read and write accesses to the PHY, while the MAC hardware implements the bitbang MDIO protocol internally. The **miiBus** code must be able to read and write to these management registers, therefore the MAC driver must export read and write methods to **miiBus**.

Many MACs must be explicitly programmed to match the link speed (10, 100, or 1000 Mb/s) and the duplex state (full or half) of the PHY in order to function correctly. A MAC driver for such a device must be notified when the link state changes, so that it can synchronize its state with that of the PHY. To do this, the MAC driver must export a link update method, **{miiLinkUpdate}( )**, so that the **miiBusMonitor** task (or the **tNet0** task) can notify the MAC driver when the link state changes.

When a MAC driver publishes the **{miiLinkUpdate}( )** method, it usually publishes the **{miiRead}( )** and **{miiWrite}( )** methods as well. These methods are always called from task context. For more information on these methods, see *6.2.2 VxBus Driver Methods for Network Interface Drivers*, p.88.

### Lower Edge Methods

The lower edge MII bus methods are exported by PHY drivers only. Currently, there are only two methods available: **{miiModeSet}( )** and **{miiModeGet}( )**. These methods are used to get and set the PHY media mode and are used internally by the **miiBusModeGet( )** and **miiBusModeSet( )** routines provided by **miiBus**. Each PHY driver must export these methods in order for the **miiBus** code to properly use the driver to manage the link.

### {miiModeSet}( )

The **{miiModeSet}( )** method is defined as follows:

```
LOCAL STATUS miiModeSet
    (
    VXB_DEVICE_ID pInst,
    UINT32 mode
    )
    {
    return (OK);
    }
```

The **{miiModeSet}( )** method is used to set the PHY link to a particular mode. The **pInst** argument is a pointer to the PHY instance context. The **mode** argument is an encoded value using the definitions from the **endMedia.h** header file. The following example illustrates how to decode the mode value to obtain the speed and duplex values:

```
switch(IFM_SUBTYPE(mode)) {
case IFM_AUTO:
    /* Autonegotiation */
    break;
case IFM_1000_SX:
    /* 1000baseSX, fiber */
    break;
case IFM_1000_T:
    /* 1000baseT, copper */
    break;
case IFM_100_TX:
    /* 100baseTX, copper */
    break;
case IFM_10_T:
    /* 10baseT, copper */
    break;
default:
    return (ERROR);
    break;
}

if ((mode & IFM_GMASK) == IFM_FDX)
    /* full duplex mode */
else
    /* half duplex mode */
```

Prior to setting the link state, the **func{miiModeSet}( )** routine should also reset the PHY and perform any required initialization. This can include applying software workarounds for hardware bugs, such as DSP fix ups. Forcing a reset typically causes a momentary link drop, which forces the link partner to also re-sense the link. This is useful for insuring that the link partner actually detects changes made to the PHY media settings.

When attempting to explicitly specify a link speed or duplex setting (rather than using autonegotiation), Wind River recommends that it be done while keeping autonegotiation enabled. While it is possible to disable autonegotiation and manually configure the PHY for a specific mode, this method can cause problems in some situations. For example, if the PHY is forced to 100 Mb/s full duplex with autonegotiation disabled, but the PHY is connected to a link partner that still has autonegotiation enabled, the link partner will use parallel detection to sense the link speed and default to half duplex. This results in a duplex mismatch that seriously degrades network performance. In addition, manual link configuration is not normally recommended for 1000 Mb/s links.

A more reliable method is to leave autonegotiation enabled, but only advertise the particular mode that is desired. For example, to force the link to 10 Mb/s full duplex, the autonegotiation advertisement register (**ANAR**) can be programed to only have the **10FD** bit set. Then, the **autoneg session restart** bit is set in the control register. This causes the current PHY and its link partner to agree that 10 Mb/s full duplex is the best common mode for the link. Tests show that this method is fairly interoperable among a wide variety of PHY devices. Consequently, this is the mechanism that the **genericPhy** driver uses.

### {miiModeGet}( )

The **{miiModeGet}( )** method is defined as follows:

```
LOCAL STATUS miiModeGet
    (
    VXB_DEVICE_ID pInst,
    UINT32 * mode,
    UINT32 * status
    )
    {
    return (OK);
    }
```

The **{miiModeGet}( )** method is used to return the current link state information. As with the **func{miiModeSet}( )** routine, the **pInst** is a pointer to the PHY instance context. The **mode** and **status** arguments point to storage where the **{miiModeGet}( )** method returns the current link settings, and the link status. The **mode** value is specified in terms of the macros defined in the **endMedia.h** header file. The **status** field sets the **IFM_AVALID** bit if it contains valid data, and the **IFM_ACTIVE** bit is set if the link is up.

**NOTE:** The **{miiModeSet}( )** method should be called at least once to initialize the PHY before the **{miiModeGet}( )** method is used to check the link state.

### 6.3.3 Header Files for PHY Drivers

The MII bus APIs and function prototypes are all defined in the **miiBus.h** header file. VxBus MAC drivers and PHY drivers should include it as follows:

```
#include <../src/hwif/h/mii/miiBus.h>
```

Individual PHY drivers may have their own header files located in the *installDir*/**vxworks-6.***x*/**target/src/hwif/h/mii** directory as well.

### 6.3.4 BSP Configuration for PHY Drivers

Because MII bus and PHY instances are autodiscovered, little BSP configuration is required. No changes to the **hwconf.c** file are normally needed. The **config.h** file should include the MII bus component and the necessary PHY drivers, as follows:

```
#define INCLUDE_PARAM_SYS
#define INCLUDE_MII_BUS
#define INCLUDE_GENERICPHY
#define INCLUDE_GENERICTBIPHY
```

### 6.3.5 Available Utility Routines for PHY Drivers

The MII bus module provides two sets of routines: upper edge routines, which are used by Ethernet MAC drivers, and lower edge routines, which are used by PHY drivers themselves. The upper edge routines are typically used to create an MII bus and manage the link. The lower edge routines are used by the PHY drivers to connect themselves to the MII bus subsystem.

#### Upper Edge Utility Routines

There are a number of upper edge utility routines available. These include: **miiBusCreate( )**, **miiBusDelete( )**, **miiBusModeGet( )**, **miiBusModeSet( )**, and **miiBusMediaListGet( )**. For more information on these routines, see *6.2.5 Available Utility Routines for Network Interface Drivers*, p.94.

### Lower Edge Utility Routines

The following lower edge utility routines are available:

#### miiBusMediaAdd( )

The **miiBusMediaAdd( )** routine is defined as follows:

```
STATUS miiBusMediaAdd
    (
    VXB_DEVICE_ID pInst,
    UINT32 media
    )
```

This routine is used by PHY drivers to notify the MII bus about the media types that they support. The routine is normally called by a PHY driver's initialization code, and is used to populate the media list information that is returned by the **miiBusMediaListGet( )** routine described in *miiBusMediaListGet( )*, p.101. A typical 10/100 Ethernet PHY might specify its media support as shown in the following example:

```
miiBusMediaAdd (pBus, IFM_ETHER|IFM_100_TX);
miiBusMediaAdd (pBus, IFM_ETHER|IFM_100_TX|IFM_FDX);
miiBusMediaAdd (pBus, IFM_ETHER|IFM_10_T);
miiBusMediaAdd (pBus, IFM_ETHER|IFM_10_T|IFM_FDX);
miiBusMediaAdd (pBus, IFM_ETHER|IFM_AUTO);
```

#### miiBusMediaDel( )

The **miiBusMediaDel( )** routine is defined as follows:

```
STATUS miiBusMediaDel
    (
    VXB_DEVICE_ID pInst,
    UINT32 media
    )
```

This routine is used by PHY drivers to remove their supported media types from their parent bus' media list when the device is unloaded.

#### miiBusMediaDefaultSet( )

The **miiBusMediaDefaultSet( )** routine is defined as follows:

```
STATUS miiBusMediaDefaultSet
    (
    VXB_DEVICE_ID pInst,
    UINT32 media
    )
```

This routine is used by PHY drivers to specify the default media selection that should be listed in the media list for a given bus. Typically, the default media type is **IFM_AUTO**.

#### miiBusRead( )

The **miiBusRead( )** routine is defined as follows:

```
STATUS miiBusRead
    (
    VXB_DEVICE_ID pInst,
    int phyAddr,
    int phyReg,
    UINT16 * regVal
    )
```

This routine is used by a PHY instance to read its own registers. The **pInst** argument is a pointer to the parent MII bus. This routine in turn invokes the **{miiRead}( )** method exported by the parent Ethernet MAC driver.

#### miiBusWrite( )

The **miiBusWrite( )** routine is defined as follows:

```
STATUS miiBusWrite
    (
    VXB_DEVICE_ID pInst,
    int phyAddr,
    int phyReg,
    UINT16 regVal
    )
```

This routine is used by a PHY instance to write its own registers. The **pInst** argument is a pointer to the parent MII bus. This routine in turn invokes the **{miiWrite}( )** method exported by the parent Ethernet MAC driver.

**NOTE:** Register values are always in native byte order.

### 6.3.6 Initialization for PHY Drivers

Any initialization that needs to be done for this driver type should be done in VxBus initialization phase 2 (**devInstanceInit2( )**).

### 6.3.7 Debugging PHY Drivers

Most problems with PHY drivers occur due to problems with the **{miiRead}( )** and **{miiWrite}( )** methods exported by the parent Ethernet MAC driver. This code can be difficult to write, particularly when serial bitbang I/O is required. When writing a new MAC driver, it is often useful to add instrumentation to the **{miiRead}( )** method to print out the results of the read access in order to see what registers are being read, and what the contents are. During normal operation, these messages are generated whenever the **miiBusMonitor** tasks invokes the **{miiRead}( )** method.



It is also useful to instrument the MAC driver **{miiLinkUpdate}( )** method in order to obtain a visual indication of when link change events are triggered.

Debugging PHY driver startup can be complicated by the fact that normally a MAC driver's initialization routines are invoked well before the system is ready to display messages on the console. Invoking the **miiBusCreate( )** routine at this time makes it difficult to observe any debug instrumentation in PHY drivers. To avoid this, Wind River recommends that you call the **miiBusCreate( )** routine from the MAC driver **{muxConnect}( )** method, because this method is always invoked as part of network initialization, well after the console device is initialized.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

## 6.4 Wireless Ethernet Drivers

Wireless Ethernet drivers do not conform to the VxBus device driver model and are not covered as part of this manual. Wind River provides 802.11 technology as part of the Wind River Wireless Ethernet Drivers product. The Wireless Ethernet Drivers product focuses mainly on drivers for the Atheros and Broadcom chipsets. For information on writing and using wireless Ethernet drivers, see the *Wind River Wireless Ethernet Drivers for VxWorks 6 User's Guide*.

## 6.5 Hierarchical END Drivers

In a previous release, Wind River introduced a model for network drivers called hEND, or Hierarchical END. The hEND model provided a mechanism for driver developers to write network interface drivers for the subset of devices that conform well to the hEND model.

The hEND model divided the END driver into two levels. The SL, or system level, which interfaced with a protocol layer, such as IP, and with the VxBus infrastructure. The DL, or device-specific level, handled all hardware-specific accesses.

The hEND model has been deprecated. Due to the cost of testing modifications to the SL, the difficulty of adapting the DL to devices that do not conform well to the model, and for better performance, current network drivers provided by Wind River do not use this model.

# 7 Non-Volatile RAM Drivers


7.1 Introduction 135

7.2 Non-Volatile RAM Drivers 136

7.3 Flash File System Support with TrueFFS 141


## 7.1 Introduction

This chapter describes non-volatile RAM (NVRAM) drivers and the VxWorks TrueFFS flash file system product. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

### NVRAM Drivers and TrueFFS

VxWorks can be configured to maintain several types of information in various types of non-volatile RAM devices. This typically includes the boot image, information used to configure the boot image (bootline), network interface hardware addresses, and flash file systems. Other kinds of information can be maintained on NVRAM devices as well.

Within the VxBus framework, NVRAM drivers are used to manage the NVRAM devices. The management tasks include allocating individual sectors to a specific purpose, writing data to sectors, and making sector data available to other parts of the system. NVRAM drivers do not maintain any information for file systems other than, possibly, allocation of space to the file system.

Wind River also provides the TrueFFS flash file system product. This is a file system support layer for use with the DosFS file system on flash devices. Other than file system support functions, TrueFFS does not manage allocation of NVRAM devices to other parts of the system.

At the time of this writing, these two mechanisms are not integrated with each other. However, both NVRAM drivers and TrueFFS are documented in this chapter. The first part of the chapter discusses NVRAM drivers, which conform to the VxBus model. The remainder of the chapter discusses TrueFFS flash file system development.

**NOTE:** TrueFFS does not conform to the VxBus device driver model.

## 7.2 Non-Volatile RAM Drivers

VxBus NVRAM drivers provide a low-level interface for allocating NVRAM sectors to other parts of the system and for reading and writing NVRAM devices.

### 7.2.1 NVRAM Driver Overview

VxBus drivers for NVRAM devices are used to allocate blocks of NVRAM for use by other drivers and modules. These drivers also provide interfaces for other drivers and modules to read and write the data contained in the blocks of NVRAM. The types of information stored in NVRAM typically include bootline information, hardware (MAC) addresses for some network interface, vendor-provided firmware, **bootrom** images, space used by applications, and so forth.

NVRAM drivers divide the available non-volatile memory into blocks for allocation to other drivers and modules. Each driver or module is identified by a pair consisting of a name string and a unit number.

### 7.2.2 VxBus Driver Methods for NVRAM Drivers

There are two VxBus driver methods used by NVRAM drivers: **{nonVolGet}( )** and **{nonVolSet}( )**.

#### {nonVolGet}( )

**{nonVolGet}( )** is called from the general-purpose routine **vxbNonVolGet( )**. The routine associated with this method, **func{nonVolGet}( )**, copies data from the appropriate block of the NVRAM device into a user-provided buffer.

```
STATUS func{nonVolGet}
    (
    VXB_DEVICE_ID       pInst,  /* VXB_DEVICE_ID of vxbFileNvRam */
    char *              drvName,/* name of requestor */
    int                 drvUnit,/* unit of requestor */
    char *              buff,   /* location to write to */
    int                 len     /* size of buff */
    )
```

#### {nonVolSet}( )

**{nonVolSet}( )** is called from the general-purpose routine **vxbNonVolSet( )**. The routine associated with this method, **func{nonVolSet}( )**, copies data from a user-provided buffer into the appropriate block of the NVRAM device.

```
STATUS func{nonVolSet}
    (
    VXB_DEVICE_ID       pInst,  /* VXB_DEVICE_ID of vxbFileNvRam */
    char *              drvName,/* name of requestor */
    int                 drvUnit,/* unit of requestor */
    char *              buff,   /* location to write to */
    int                 len     /* size of buff */
    )
```

### 7.2.3 Header Files

Only one driver-class specific header file is used for NVRAM drivers. This is the **vxbNonVol.h** header:

```
#include <hwif/util/vxbNonVol.h>
```

### 7.2.4 BSP Configuration for NVRAM Drivers

When non-volatile RAM drivers are used, the BSP should be configured to include **INCLUDE_NON_VOLATILE_RAM**.

```
#define INCLUDE_NON_VOLATILE_RAM
```

To configure individual NVRAM drivers, there are two resource names used: **segments** and **numSegments**. The **segments** resource points to the beginning of a table containing information about each segment of the NVRAM device. The table consists of records of the **struct nvRamSegment** type. The **nvRamSegment** structure contains four fields:

**segAddr**
Indicates the address of the segment as a byte offset from the beginning of the NVRAM device.

**segSize**
Indicates the size of the segment in bytes.

**name**
Indicates the name of the driver or other module to which the segment is allocated.

**unit**
Indicates the unit number of the driver or other module to which the segment is allocated.

For example:

```
typedef struct nvRamSegment NVRAM_SEGMENT;
struct nvRamSegment
    {
    void *        segAddr;
    int           segSize;
    char *        name;
    int           unit;
    };
```

The following is a sample of the **hwconf.c** record for the NVRAM of a hypothetical D1643 device.

```
const struct nvRamSegment d16430Segments[] = {
    /* IBM Eval kit software use */
    { 0, 1024, "IBMEvalKit", 0 },

    /* bootline */
    { NV_BOOT_OFFSET, BOOT_LINE_SIZE, "BOOTLINE", 0 },
```

```
    /* emac 0 and 1 */
    { (NV_BOOT_OFFSET + NV_ENET_OFFSET_0), 6, "emac" 0 },
    { (NV_BOOT_OFFSET + NV_ENET_OFFSET_1), 6, "emac" 1 },
};

const struct hcfResource d16430Resources[] = {
    { "regBase",    HCF_RES_INT, { (void *)NV_RAM_ADRS } },
    { "segments",   HCF_RES_ADDR, { (void *)&d16430Segments[0] } },
    { "numSegments",HCF_RES_INT, { (void *)NELEMENTS(d16430Segments } },
};
#define d16430Num NELEMENTS(d16430Resources)
```

As for every resource table, the **hcfDeviceList[ ]** table must have an entry for the specific resource table:

```
{ "d1643", 0, VXB_BUSID_PLB, 0, d1643Num, d1643Resources },
```



### 7.2.5 Utility Routines for NVRAM Drivers

There are no general-purpose utility routines required for NVRAM drivers.

NVRAM drivers get a pointer to the head of the **nvRamSegment** table, and the size of the table, using **devResourceGet( )**. For example:

```
devResourceGet(pHcf, "segments", HCF_RES_ADDR, (void*)&(pDrvCtrl->segTable));
devResourceGet(pHcf, "numSegments", HCF_RES_INT,
              (void*)&(pDrvCtrl->segTblSize));
```

### 7.2.6 Initialization for NVRAM Drivers

In most cases, NVRAM drivers perform all initialization in the first phase of VxBus initialization (**devInstanceInit( )**). As a service, NVRAM drivers sometimes provide NVRAM information contents to other drivers. This includes items such as network device hardware addresses. Other drivers require this information during their own phase 2 initialization routines. For this reason, NVRAM drivers generally complete their initialization during VxBus phase 1 initialization.

### 7.2.7 NVRAM Block Sizes

The size and arrangement of the NVRAM blocks is usually determined by the hardware. With flash parts, each block consists of a single erase unit. For example, a single Am29LV320D flash part provides eight 8 KB erase units and 63 64 KB erase units, where each erase unit can be treated by the driver as a separately allocatable block.

Battery-backed RAM is an exception to this rule. Typically, each byte of battery-backed RAM can be separately written, therefore the block sizes of these devices can be set to any number of bytes.

A single block must be allocated to exactly one driver or other module. Do not attempt to split a block into smaller allocations as this can result in large system overhead. For example, if a network hardware address needs to be stored on the Am29LV320D flash part described previously, the BSP must allocate at least one 8 KB erase unit to store the six bytes of information required.

If NVRAM flash storage is at a premium in your system, no battery-backed RAM is available, and RAM is readily available, see *7.2.8 Stacking NVRAM Instances*, p.140 for an alternative allocation method.

While the device typically determines the sizes and layout of the blocks, the BSP determines what each block is allocated to. From the perspective of the BSP, a single allocation can cover more than one block. Your driver must be able to recognize and handle this situation.

**NOTE:** Once the NVRAM allocations are set, they must be maintained in the same places. Changing the locations of NVRAM allocations results in corrupt data unless the NVRAM device is erased and completely rewritten with the new organization.

### 7.2.8 Stacking NVRAM Instances

In some situations, it is possible to write a shim NVRAM driver that does not manage any physical hardware. Instead, the shim driver allocates one or more blocks on some other NVRAM device.

The shim driver provides arbitrary sized block allocations for drivers and other modules. This means that the shim driver reads the flash into RAM and maintains the contents in RAM. At a time that is appropriate for the application, the shim driver writes the contents back to the flash segments using **vxbNonVolSet( )**. For more information on **vxbNonVolSet( )**, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.

### 7.2.9 Debugging NVRAM Drivers

During early stages of system initialization, NVRAM drivers are not typically required in order for the system to boot and for devices such as the console to work. Therefore, no special debugging requirements exist.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.



## 7.3 Flash File System Support with TrueFFS

TrueFFS is an optional product that allows a file system to be used and maintained on flash media. The TrueFFS module is not a driver in the traditional sense and is not integrated with the VxBus driver model. 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*.

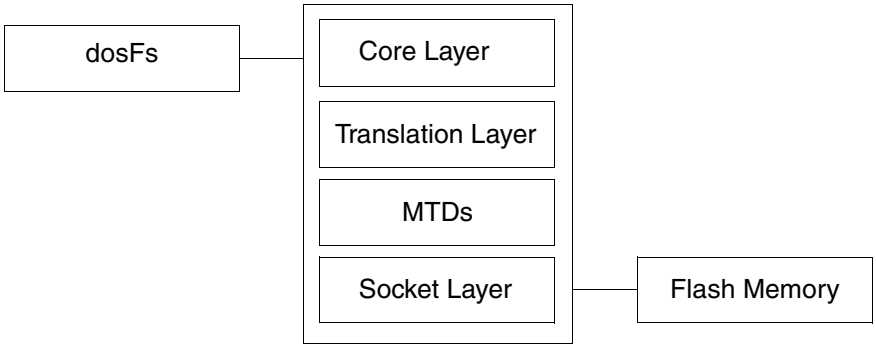
This chapter contains information necessary to write routines for TrueFFS support of new devices.

### 7.3.1 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 7-9.

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 7-1. The three functional layers are provided in source code form, in binary form, or in both, as noted in the following sections.

Figure 7-1 **TrueFFS Layers**




### 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.

### 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.

### 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.

#### 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.

### 7.3.2 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*.

#### 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.145.

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_MODE_REQUIRES_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 **sysTffs.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*).

### 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:

```
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:

```
vol.erasableBlockSize = aValue * vol.interleaving;
```

Where *aValue* is the erasable block size of a flash chip that is not interleaved with another.

**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 **flFitInSocketWindow( )** global routine.

**noOfChips**
The number of flash memory chips used to construct the flash memory array.

**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.

**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.

**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 **mtdVars** are used for the variables that are instance-specific, so that an MTD may be used more than once in a system.

**socket**
This member is a pointer to the **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 *Socket Drivers*, p.157). Further, because TrueFFS uses these socket driver routines to access the flash memory hardware, you must register your socket driver *before* you try to run the MTD identify routine that initializes the bulk of this structure.

**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:

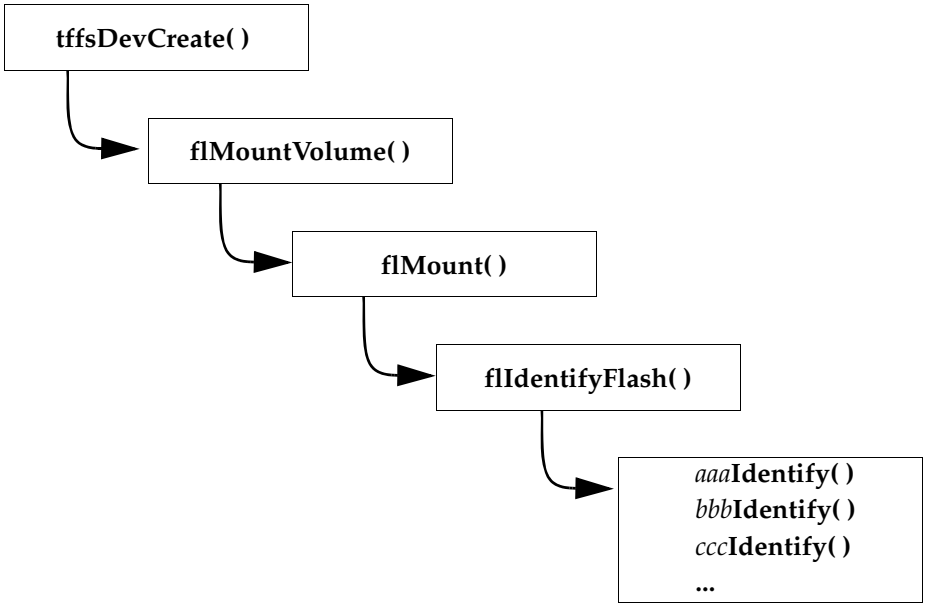
```
FLStatus myMTDIdentification
    (
    FLFlash vol
    )
    {
    /* Do what is needed for identification */

    /* If identification fails return appropriate error */

    return flOK;
    }
```

After setting the members listed above, this routine should return **flOK**.

#### Call Sequence

Upon success, the identification routine updates the **FLFlash** structure, which also completes the initialization of the **FLSocket** structure referenced within this **FLFlash** structure.

Figure 7-2 **Identifying an MTD for the Flash Technology**




### 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 **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.

#### 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.

1. An MTD does not need to touch Vcc. TrueFFS turns Vcc on before calling an MTD 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** because it is only called from the volume descriptor. The stub provided is:

```
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:

```
Static FLStatus myMTDErase
    (
    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:

```
Component INCLUDE_MTD_type
    {
    NAME            name
    SYNOPSIS        type devices
    MODULES         filename.o
    HDR_FILES       tffs/flflash.h tffs/backdrnd.h
    REQUIRES        INCLUDE_TFFS \
                    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:

```
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:

```
#ifdef INCLUDE_MTD_USR
    usrMTDIdenitfy,
#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.

### 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 *xxx***Register( )** 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:

   ```
   #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 Registration 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 *xxx***Register( )** call in pre-processor conditional statements, as in the following example:

```
#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.

**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.

**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.

**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:

```
FLSocket vol = flSocketOf (noOfDrives);
```

Here, **noOfDrives** is the running count of drives attached to the system. The routine **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.[2] 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 **cardChanged** to **TRUE**.

- **rfaVccOn( )**

TrueFFS can call this routine to turn on *Vcc*, which is the operating voltage. For the flash memory hardware, Vcc is usually either 5 or 3.3 Volts. When the media is idle, TrueFFS conserves power by turning 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 Vcc-off mechanism is used, in which Vcc is turned off only after at least one interval has passed. If several flash-accessing operations are executed in rapid sequence, 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 Vcc off. However, when socket polling is active, Vcc is turned off only after a delay. Thus, if several flags accessing operations are executed in rapid sequence, Vcc is left on during the sequence. Vcc is turned off only when TrueFFS goes into an idle state. 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.

2. TrueFFS only supports a maximum of 5 drives numbered 0-4.

**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 *xxx***SetWindow( )** and *xxx***SetMappingContext( )** 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.

## Flash Translation Layer

This section provides a detailed discussion of the *flash translation layer* (FTL).

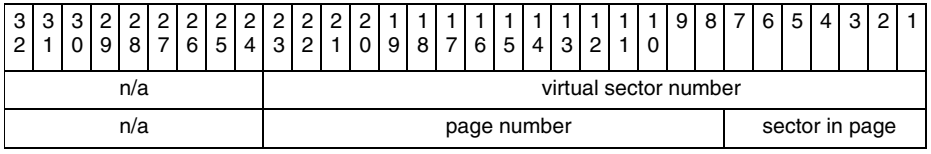
### Terminology

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.

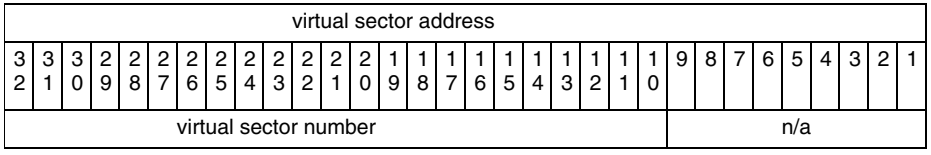
Figure 7-3 **Virtual Sector Number**




*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.

Figure 7-4 **Virtual Sector Address**



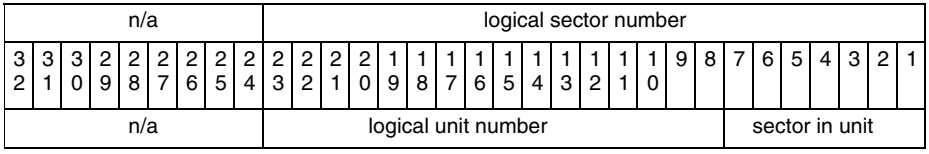

*virtual sector*

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.

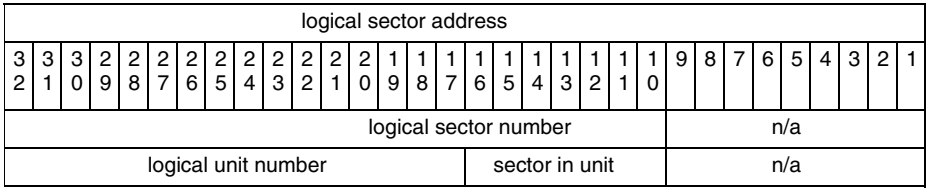
Figure 7-5 **Logical Sector Number**




*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.

Figure 7-6 **Logical Sector Address**




*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.

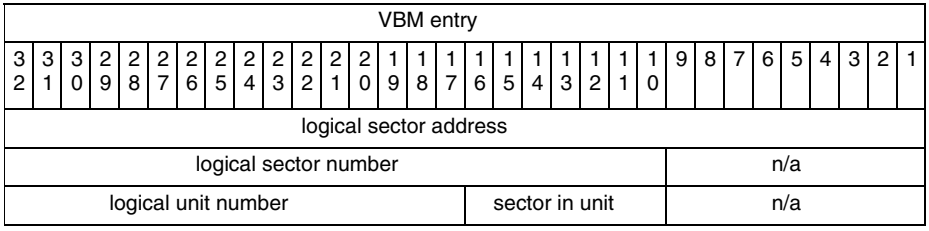
*physical erase unit*
This is the same as an *erase unit*.

*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.

Figure 7-7 **Virtual Block Map (VBM)**

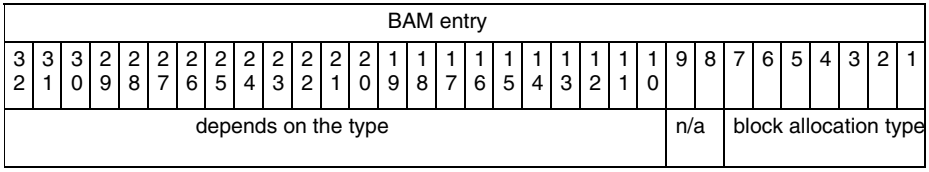





*block allocation map*

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.

Figure 7-8 **Block Allocation Map (BAM)**




*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.177.) 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 **typedef**s 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 tTLrec {
  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           unitOffsetMask;          /* = 1 << unitSizeBits - 1 */
  CardAddress           bamOffset;
  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 */
                                                  /* directly addressable sectors */
  LogicalSectorNo       replacementPageAddress;
  VirtualSectorNo       replacementPageNo;
  SectorNo              mappedSectorNo;
  const void FAR0 *     mappedSector;
  CardAddress           mappedSectorAddress;
  unsigned long         currWearLevelingInfo;
#ifdef BACKGROUND
  Unit *                unitEraseInProgress;      /* Unit currently being formatted */
  FLStatus              garbageCollectStatus;     /* Status of garbage collection
  /*
   * When unit transfer is in the background, and is currently in progress,
   * all write operations done on the 'from' unit must be mirrored on the
   * transfer unit. If so, 'mirrorOffset' will be non-zero and is the
   * offset of the alternate address from the original. 'mirrorFrom' and
   * 'mirrorTo' are the limits of the original addresses to mirror.
   */
  long int              mirrorOffset;
  CardAddress           mirrorFrom, mirrorTo;
#endif
#ifndef SINGLE_BUFFER
  FLBuffer *            volBuffer;                /* Define a sector buffer */
#endif
  FLFlash               flash;
#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.

```
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(**vol.flash.erasableBlockSize**) / ln(2)).

```
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(**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.

```
noOfUnits = ((vol.flash.noOfChips * vol.flash.chipSize) >>
             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.

```
firstPhysicalEUN  = (((formatParams->bootImageLen - 1) >>
                      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.

```
noOfPages = ((vol.virtualSectors * SECTOR_SIZE - 1) >>
              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**.

```
directAddressingSectors = (formatParams->vmAddressingLimit /
                           SECTOR_SIZE) + 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**.

```
directAddressingMemory = 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.

```
unitOffsetMask = (1L << 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.

```
bamOffset = sizeof(UnitHeader);
```

or this if **formatParams->embeddedCISlength** is not 0

```
bamOffset = sizeof(UnitHeader) – (sizeof uh->embeddedCIS) +
            (formatParams->embeddedCISlength + 3) / 4 * 4;
```

**sectorsPerUnit**

**sectorsPerUnit** is the number of physical sectors in an erase unit.

```
sectorsPerUnit = (1 << (vol.unitSizeBits - 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.

```
unitHeaderSectors = ((allocEntryOffset(&vol,vol.sectorsPerUnit) - 1) >>
                     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

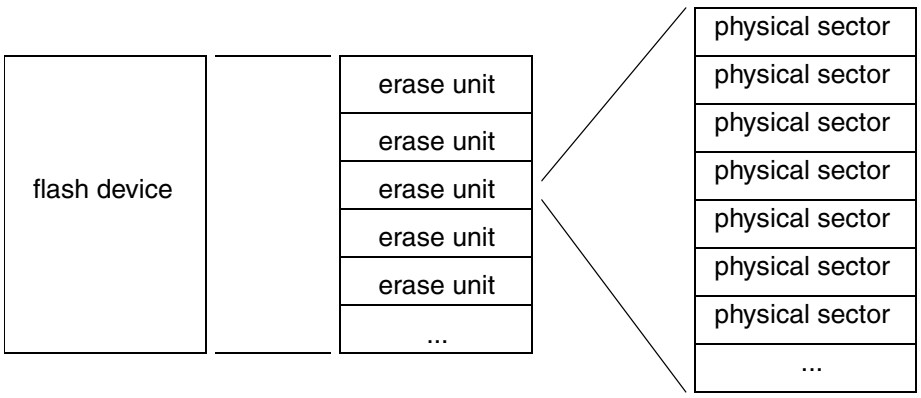
Flash chips are sometimes interleaved to allow larger word access to flash. That is, two 8-bit flash chips can be interleaved together to allow 16-bit access. When working with interleaved flash, the smallest area that can be erased is increased. This smallest erasable area is called an erase unit.

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.

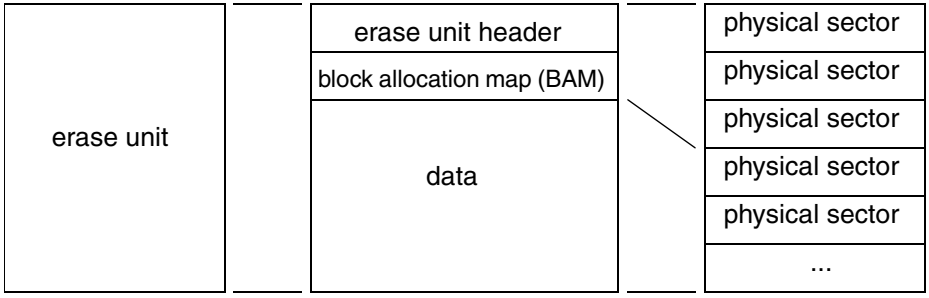
**NOTE:** As a note, erasing an erase unit is sometimes called formatting. This is not the same as a DOS format.

Figure 7-9 **Flash Device Layout**




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.

Figure 7-10 **Erase Unit Layout**






### 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.

**NOTE:** All values in the unit header are little-endian, thus care is required when reading or writing to this table.

```
typedef struct
    {
    char            formatPattern[15];
    unsigned char   noOfTransferUnits;      /* no. of transfer units */
    LEulong         wearLevelingInfo;
    LEushort        logicalUnitNo;
    unsigned char   log2SectorSize;
    unsigned char   log2UnitSize;
    LEushort        firstPhysicalEUN;       /* units reserved for boot
                                               image */
    LEushort        noOfUnits;              /* no. of formatted units */
    LEulong         virtualMediumSize;      /* virtual size of volume */
    LEulong         directAddressingMemory; /* directly addressable memory */
    LEushort        noOfPages;              /* no. of virtual pages */
    unsigned char   flags;
    unsigned char   eccCode;
    LEulong         serialNumber;
    LEulong         altEUHoffset;
    LEulong         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 **noOfTranferUnits** 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.physcialUnits[ ]** 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.

```
#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)).

```
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);
firstPhysicalEUN = vol.firstPhysicalEUN;
```

**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);
noOfUnits =(vol.noOfUnits - vol.firstPhysicalEUN);
```

**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.sectorPerUnit - 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.

```
vol.bamOffset = sizeof(UnitHeader);
```

or this if **formatParams->embeddedCISlength** is not 0

```
vol.bamOffset = sizeof(UnitHeader) – (sizeof uh->embeddedCIS) +
                (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**.

# 8 Resource Drivers


8.1 Introduction 185

8.2 Overview 186

8.3 VxBus Driver Methods 186

8.4 Header Files 187

8.5 BSP Configuration 187

8.6 Available Utility Routines 187

8.7 Initialization 187

8.8 Debugging 188


## 8.1 Introduction

This chapter describes resource drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 8.2 Overview

Many target systems include special purpose resources that can be allocated in one of several ways for use by other devices and drivers. Special purpose resources are typically available when there is a resource in the system that can be used by multiple devices, when a resource is expensive, or when there are more consumer devices for the resource than there are resources available. These resources include hardware elements such as switching and routing systems to transfer bus traffic from a bus controller to the bus it manages.

Management of these resources should be done by a resource driver dedicated to managing the resource. The management can involve allocation of the resource to other drivers, startup configuration, or run-time management of a limited resource.

As a rule, resource drivers are highly custom. You can think of these drivers as the glue code that makes the system work correctly.

## 8.3 VxBus Driver Methods

There are generally two operations that resource drivers perform. First, the driver can allocate some resource. Second, the driver can manage a resource on behalf of some other entity on the system, which may or may not involve allocation to the requestor.

Wind River provides custom methods for resource drivers, for example, **{cpmCommand}( )** and **{m85xxLawBarAlloc}( )**. These custom methods should not be advertised by new drivers, as they are custom-defined for the drivers that already use them, and creating new drivers that advertise these methods causes an adverse impact for other drivers.

The only method that can be used by developers of third-party resource drivers is the generic **{driverControl}( )** method. This method is described in *13. Other Driver Classes*.

## 8.4 Header Files

There are no header files that are applicable to resource drivers as a class. However, when a resource driver is written specifically for a single chip, there may be header files shared by the devices on that chip.

## 8.5 BSP Configuration



Because resource drivers are highly customized, there can be custom BSP requirements for a resource driver. In some cases, this involves allowing the BSP to configure the allocation of resources to drivers that require use of the resources.

For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.

## 8.6 Available Utility Routines

There are no utility routines available that are specific to resource drivers.

## 8.7 Initialization

Other drivers in the system may be dependent on resource drivers and may have initialization restrictions of their own. For this reason, resource drivers are often required to complete their initialization in VxBus initialization phase 1 (**devInstanceInit( )**).

## 8.8 Debugging

For most development, resource drivers are simple to debug because the system can be completely up before the resource driver is loaded. However, if the system cannot be booted without the resource driver, or if the resource driver is required in order for all peripheral devices to be available for use as a debug interface, then debugging resource drivers can be more complex.

One way of handling this situation is to develop the resource driver as the BSP is being developed. In this case, the BSP developer may be able to hard-wire the resource allocation during initial development. The system can then be booted to a point where normal debug tools are available, and the resource driver completed at that point. To maximize your ability to reuse a given resource driver, be sure to restructure the BSP and resource driver so that the resource driver, and not the BSP, allocates the resources in the deployed system.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 9 Serial Drivers


9.1 Introduction 190
9.2 Overview 190
9.3 VxBus Driver Methods 190
9.4 Header Files 192
9.5 BSP Configuration 193
9.6 Available Utility Routines 193
9.7 Initialization 193
9.8 Polled Mode Versus Interrupt-Driven Mode 194
9.9 SIO_CHAN and SIO_DRV_FUNCS 194
9.10 WDB 197
9.11 Serial Drivers, Initialization, and Interrupts 198
9.12 Debugging 199

## 9.1 Introduction

This chapter describes VxBus serial drivers that support RS232, RS422, and other similar devices. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 9.2 Overview

VxBus serial drivers provide an interface to the I/O system similar to the pre-VxBus multi-mode (SIO) serial drivers, but also provide an enhanced initialization mechanism and a simplified interface to the BSP.

These drivers provide an interface for setting hardware options, such as the number of start bits, stop bits, data bits, parity, and so on. In addition, they provide an interface for polled communication that can provide external mode debugging (such as ROM-monitor style debugging) over a serial line.

Each serial port is associated with an **SIO_CHAN** structure. When a single device supports more than one port, the device must be able to provide a separate **SIO_CHAN** structure for each supported port. The contents of the **SIO_CHAN** structure are defined in *installDir*/**vxworks-6.***x*/**target/h/sioLib.h**. Many drivers append additional fields to the end of the **SIO_CHAN** structure, to contain additional information required by the driver.

## 9.3 VxBus Driver Methods

There are two VxBus driver methods used by serial drivers: **{sioChanGet}( )** and **{sioChanConnect}( )**. Both **func{sioChanGet}( )** and **func{sioChanConnect}( )** use

an **SIO_CHANNEL_INFO** structure as their second argument. This structure contains two pieces of information:

**sioChanNo**
An integer value indicating which channel on the system to use.

This corresponds to the number *N* in the channel name used by the I/O system: **/tyCo/***N*. That is, **0** for **/tyCo/0** or **3** for **/tyCo/3**.

**pChan**
Holds the **SIO_CHAN** structure pointer discussed in *9.9 SIO_CHAN and SIO_DRV_FUNCS*, p.194.

### 9.3.1 {sioChanGet}( )



The **func{sioChanGet}( )** routine retrieves the **SIO_CHAN** structure associated with the specified serial port.

```
void func{sioChanGet}
    (
    VXB_DEVICE_ID pDev,
    SIO_CHANNEL_INFO * pInfo
    )
```

The **SIO_CHANNEL_INFO** structure contains fields to specify a port and to return a pointer to the **SIO_CHAN** structure associated with the port. If the instance manages the specified port, the **func{sioChanGet}( )** routine must assign **pInfo->pChan** with a pointer to the **SIO_CHAN** structure associated with that port.

**NOTE:** The port specified in the **SIO_CHANNEL_INFO** structure is identified by the channel number of the serial port on the entire system. Do not assume that the identification is related only to your device. For example, if your device supports four serial ports, and the first port on your device corresponds to **/tyCo/2**, then the **pChan->channelNo** field values of **0** and **1** are not intended for your device, and the values of **2** through **5** correspond to the 0, 1, 2, and 3 ports on your device.

### 9.3.2 {sioChanConnect}( )

This driver method is used to connect the specified channel number to the I/O subsystem. There are two alternate forms of this driver method, depending on the channel selected in the **SIO_CHANNEL_INFO** structure specified as the argument to **func{sioChanConnect}( )**.

In the first form, a single channel is specified. In this case, the specified channel is connected to the I/O subsystem.

In the second form, the channel value is specified as **-1**. This value indicates that the driver should connect all channels associated with this instance.

**NOTE:** Recall that an instance is the pairing between a driver and a single device. However, in some cases, a device such as a PCI card may include multiple serial ports. In this case, there are multiple ports for a single instance. Do not confuse this with multiple instances. Only the ports associated with the specified instance should be connected when the **func{sioChanConnect}( )** driver method routine is called.

The prototype for the **func{sioChanConnect}( )** driver method is:

```
LOCAL void func{sioChanConnect}
    (
    VXB_DEVICE_ID pInst,
    void * pArg
    )
    {
    SIO_CHANNEL_INFO * pInfo = (SIO_CHANNEL_INFO *)pArg;
    ...
    }
```

This routine does not return any value, either directly or through the **SIO_CHANNEL_INFO** structure.

## 9.4 Header Files

VxBus serial drivers should include the following serial driver header files, in addition to the generic VxBus and other system header files as well as any driver-specific header files.

```
#include <sioLib.h>
#include <hwif/util/sioChanUtil.h>
```

## 9.5 BSP Configuration

Serial drivers do not typically require configuration information from a BSP that is above and beyond the normal device-specific information provided for all drivers. For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.

## 9.6 Available Utility Routines



There are no class-specific utility routines required for serial drivers. However, some of the inline macros defined in *installDir*/**vxworks-6.***x*/**target/h/sioLib.h** may be useful for your development. For a complete list of available routines, see the **sioLib.h** file.

## 9.7 Initialization

There are two primary consumers of serial devices, both of which impose initialization constraints.

When phase 1 initialization is complete, VxWorks chooses a serial port to be used as a console. Your driver must complete its initialization during phase 1 in order to be used as a console.

**NOTE:** The serial port is not used for input or output until the end of phase 2 initialization.

Serial ports can also be used for WDB connections for system-mode debugging. The serial port that is used for this purpose is selected after phase 1 initialization is complete. When used for system-mode debugging, polled-mode input and output are required before phase 2 initialization begins.

During initialization, the device should be configured in interrupt mode unless explicitly set to polled-mode using **SIO_MODE_SET** as specified in *9.9 SIO_CHAN and SIO_DRV_FUNCS*, p.194.

For additional information on serial driver initialization, see *9.11 Serial Drivers, Initialization, and Interrupts*, p.198.

## 9.8 Polled Mode Versus Interrupt-Driven Mode

VxWorks serial drivers must provide an interrupt-driven mode for normal operation. Serial drivers can also provide a polled mode for use with WDB, polled mode console output, and other polled operations. However, although support for polled mode is encouraged, it is not required.

Several of the remaining sections in this chapter contain information about the requirements and implications of providing polled mode support.

## 9.9 SIO_CHAN and SIO_DRV_FUNCS

Every SIO port is associated with an **SIO_CHAN** structure. When an instance only provides one port, serial drivers typically put the **SIO_CHAN** structure at the beginning of the driver-specific data structure **pDrvCtrl**. This allows the **pDrvCtrl** structure pointer to be identical to the **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 follows:

```
typedef struct sio_drv_funcs SIO_DRV_FUNCS;

typedef struct sio_chan         /* a serial channel */
    {
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
    } SIO_CHAN;
```

```
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. This routine provides the primary control interface for the driver. To access the I/O control services for a standard SIO device, use the following symbolic constants:

**SIO_BAUD_SET**, **SIO_BAUD_GET**

Sets and retrieves the port baud rate.

**SIO_HW_OPTS_SET**, **SIO_HW_OPTS_GET**

Sets and retrieves the port hardware options. The available options are: **CLOCAL**, **HUPCL**, **CREAD**, **CSIZE**, **PARENB**, and **PARODD**.

For more information on these options, see *installDir*/**vxworks-6.***x*/**target/h/sioLibCommon.h**.

**SIO_MODE_SET, SIO_MODE_GET, SIO_AVAIL_MODES_GET**
Sets and retrieves the port mode to switch between polled mode and interrupt driven mode, and find which modes are available. Polled mode is specified as **SIO_MODE_POLL** and interrupt driven mode is specified with **SIO_MODE_INT**. When **SIO_AVAIL_MODES_GET** is used, the values of **SIO_MODE_POLL** and **SIO_MODE_INT** are logically or-d together as follows:

```
*(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
```

**SIO_OPEN**
Sets modem control lines (**RTS** and **DTR**) to **TRUE** if not already set, and initializes the device for user operation. Only valid if **SIO_HUP** is supported.

**SIO_HUP**
Resets **RTS** and **DTR** signals.

Other **ioctl( )** commands can be supported as well. For a more complete list of **ioctl( )** commands that can be supported by serial drivers (such as keyboard modes and keyboard LED states), see *installDir*/**vxworks-6.***x*/**target/h/sioLibCommon.h**.

**txStartup( )**
Provides a pointer to the routine that the system calls when new data is available for transmission. Typically, this routine is called only from the **ttyDrv.o** module. This module provides a level of functionality that allows a raw serial channel to behave with line control and canonical character processing.

**callbackInstall( )**
Provides the driver with pointers to callback routines that the driver can call asynchronously to handle character puts and gets. The driver is responsible for saving the callback routines and arguments that it receives from the **callbackInstall( )** routine. The available callbacks are **SIO_CALLBACK_GET_TX_CHAR** and **SIO_CALLBACK_PUT_RCV_CHAR**.

- Define **SIO_CALLBACK_GET_TX_CHAR** to point to a routine that fetches a new character for output. The driver calls this callback routine with the supplied argument and an additional argument that is the address to receive the new output character (if any). The called routine returns **OK** to indicate that a character was delivered, or **ERROR** to indicate that no more characters are available.
- Define **SIO_CALLBACK_PUT_RCV_CHAR** to point to a routine the driver can use to pass characters to the system. For each incoming character, the

callback routine is called with the supplied argument, and the new character as a second argument. Drivers normally do not care about the return value from this call. In most cases, there is nothing that a driver can do but drop a character if the I/O system is not able to receive it.

**pollInput( )** and **pollOutput( )**
Provide an interface to polled mode operations of the driver. These routines are not called unless the device has already been placed into polled mode by an **SIO_MODE_SET** operation.



## 9.10 WDB

WDB can be configured to use a serial port for communication between the host and target by specifying **WDB_COMM_TYPE** with the value **WDB_COMM_SERIAL**. The primary impact of this on serial driver development is that WDB uses polled mode for this communication, therefore a driver without polled mode cannot support this configuration.

### 9.10.1 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 *VxWorks BSP Developer's Guide: Porting a BSP to Custom Hardware*). In this case, the WDB target agent issues an **ioctl( )** with **SIO_MODE_SET** as the command in order to set the device into polled mode. Later, the agent puts the driver back into normal interrupt mode. For more information on WDB, see the *VxWorks Kernel Programmer's Guide: Kernel*.

## 9.11 Serial Drivers, Initialization, and Interrupts

There are several issues related to initialization and interrupts that are particular to serial drivers.

### 9.11.1 WDB and Interrupts

As a serial driver developer, you must be aware of interactions between serial ports and 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—the only debug mode available during system bringup—WDB puts the serial port into polled mode. However, at other times, WDB puts the serial port into normal operation, which usually implies an interrupt-driven mode.

If WDB places the driver into polled mode during system bringup, then later switches to interrupt driven mode, the driver may not have a chance to attach an ISR to the device interrupt. To avoid a stray interrupt—which can cause serious problems with the system—your driver must ensure that the switch from polled mode to interrupt mode does not assume that instance initialization is complete.

Connecting an interrupt requires that the system memory pool be available. However, during the early phases of system initialization, the system memory pool is not available. Your driver must wait until the second phase of VxBus initialization, **devInstanceInit2( )**, before it can successfully connect an ISR to the device interrupt. Your driver must not switch from polled mode to interrupt mode until this initialization is complete.

The normal calling sequence is as follows:

1. **usrRoot( )**
2. **sysClkConnect( )**
3. **sysHwInit2( )**
4. **vxbDevInit( )**
5. the driver's **devInstanceInit2( )** routine
6. **vxbIntConnect( )**

If the driver attempts to connect an ISR before **usrRoot( )** runs, the attempt fails. Any subsequent interrupts are stray interrupts. These stray interrupts cause problems during system initialization.

### 9.11.2 Initialization Order and Interrupts

Another issue for serial driver developers is related to the behavior of the actual driver if it attempts to connect interrupts before the kernel is started. When this happens, the SIO driver sometimes loses the ability to function in interrupt mode thereafter, though it generally continues to work in polled mode. This is because the system is likely to overwrite any interrupt connectivity information written before the driver's **devInstanceInit2( )** routine. Alternatively, the system can crash during bootup, due to attempts to configure interrupt connectivity before the interrupt subsystem is initialized. As mentioned previously, interrupts cannot be connected before the kernel is started.



### 9.11.3 Initialization Order

For various reasons, VxBus serial drivers must perform the majority of their required initialization in the first phase of the VxBus initialization sequence. This makes them available—in polled mode—to WDB, polled mode console output, and other operations before the I/O system is available. The only initialization that required for serial drivers after the first initialization phase is connection and enabling of interrupts.

## 9.12 Debugging

When debugging a serial driver, as with all driver development, it is often most convenient to have a fully functional system to test the driver on. This allows the driver developer full access to the debug capabilities of VxWorks and the VxBus show routines, which are helpful when debugging.

This is relatively simple to accomplish for serial drivers, if you are able to develop on a target hardware platform with working PCI. When PCI support is available, you can use one of the PCI serial cards supported by the ns16550 serial driver. You

can then change the console to the PCI serial card. This provides you with full access to the VxWorks system when you start debugging your custom serial driver.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 10 Storage Drivers


10.1 Introduction 201

10.2 Overview 202

10.3 VxBus Driver Methods 202

10.4 Header Files 202

10.5 BSP Configuration 203

10.6 Available Utility Routines 203

10.7 Initialization 204

10.8 Interface with VxWorks File Systems 205

10.9 Writing New Storage Drivers 210

10.10 Debugging 211


## 10.1 Introduction

This chapter describes storage drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 10.2 Overview

The VxBus storage driver class currently encompasses parallel and serial ATA (SATA) drivers. These drivers pair with ATA or SATA host controllers to form VxBus instances. While a host controller can have multiple devices connected to it, only the host controller is a VxBus instance.

As in previous VxWorks releases, a monolithic approach has been taken in developing these drivers. Each driver is responsible for providing block device management routines, spawning device monitoring and job handling tasks, and performing low-level device access.

VxBus storage drivers provide block device management routines so that various VxWorks file systems can be mounted on the connected device(s). Device monitoring, such as handling connect and disconnect events and interrupts, must also be provided by the storage driver.

## 10.3 VxBus Driver Methods

VxBus storage drivers do not use or supply any VxBus driver methods. During VxBus initialization phase 3 (**devInstanceConnect( )**), the driver initializes the controller device and sets up the ability to recognize storage media and announce the media to the XBD block device abstraction layer described in *10.8 Interface with VxWorks File Systems*, p.205 and in *10.7 Initialization*, p.204.

## 10.4 Header Files

Although there are no class-specific header files for storage drivers, each storage driver must include the following header files in order to use the higher-level block device and event handling utilities.

```
#include <drv/xbd/xbd.h>
#include <drv/erf/erfLib.h>
#include <drv/xbd/bio.h>
```

## 10.5 BSP Configuration

Storage drivers do not typically require configuration information from a BSP that is above and beyond the normal device-specific information provided for all drivers.

There are two storage-class specific structures required for this class: **ATA_RESOURCE** and **ATA_TYPE**. Some drivers require these structures to be provided by the BSP. If this is the case for your driver, use **ataResources** and **ataTypes** as the resource names.

For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.



## 10.6 Available Utility Routines

This section briefly describes the utility routines available for storage class drivers. These routines are discussed further in *10.8 Interface with VxWorks File Systems*, p.205.

#### erfHandlerRegister( ) and erfHandlerUnregister( )

The **erfHandlerRegister( )** routine registers a routine with the error reporting framework (ERF) that is called when the XBD is fully initialized. **erfHandlerUnregister( )** de-registers the routine registered by **erfHandlerRegister( )**.

For more information on these routines, see *ERF Registration*, p.205.

#### erfEventRaise( )

The **erfEventRaise( )** routine announces to the system that a new device has been added and that it is ready for file system mounting.

For more information on this routine, see *10.8.3 Event Reporting*, p.208.

#### xbdAttach( )

The **xbdAttach( )** routine advertises the **xbd_funcs** structure to the system. The **xbd_funcs** structure provides the following routines:

- **(*xf_ioctl)( )**—provides a single interface to various driver functions such as device eject.
- **(*xf_strategy)( )**—queues writes and reads to a given storage device (see *10.8.2 Processing*, p.208).
- **(*xf_dump)( )**—allows the driver to provide a method for writing data to a device in the event of a system failure.

For more information, see *Advertisement of XBD Methods*, p.206.

#### bio_done( )

The **bio_done( )** routine is used to mark the **bio** structure as processed. For more information, see *10.8.2 Processing*, p.208.

## 10.7 Initialization

Because storage drivers depend on the XBD library and other parts of the VxWorks I/O system, they must normally wait until these services are initialized before they are initialized. This implies that the bulk of the storage driver initialization must be done in the VxBus initialization phase 3 (**devInstanceConnect( )**).

Generally, any initialization tasks that make calls—or may potentially lead to calls—to the XBD or ERF libraries must be done no earlier than VxBus initialization phase 3. The is most obvious in the case where device creation is triggered by a device change interrupt.

# 10.8 Interface with VxWorks File Systems

The storage class drivers utilize two interrelated VxWorks subsystem libraries: eXtended Block Device (XBD) and Event Reporting Framework (ERF). These libraries facilitate the interface between the device drivers and the VxWorks file systems. This section discusses how storage class drivers should use these libraries in different areas of operation.

## 10.8.1 Device Creation

Storage class drivers typically provide a routine that creates the XBD block device structures and reports to the XBD layer when the underlying device is ready to be used. There are two ways to implement this functionality. The Intel ICH driver presents one implementation. This driver is configured to call this routine during system initialization, based on data contained in the BSP **hwconf.c** file. The Silicon Image driver uses another implementation. This driver uses a port monitoring task to dynamically create block device structures upon device detection. The preferred method for new development is the implementation found in the Silicon Image driver.



The creation routine, regardless of how it is called, is responsible for the following initialization duties:

- Allocating and initializing the block device structure.
- Spawning the device service task.
- Initializing semaphore(s) needed for task synchronization.
- Registering with the ERF.
- Advertising XBD methods.
- Notifying the ERF of a new device.

The last three items are discussed further in the following sections.

#### ERF Registration

The ERF provides a means for notifying the storage driver that the XBD initialization for the device being created is complete. Your driver should use the

routine **erfHandlerRegister( )** to register a routine with the ERF. This routine is then called when the XBD is fully initialized. For example:

```
erfHandlerRegister(xbdEventCategory, xbdEventInstantiated,
                   myDeviceXbdReadyHandler, pMyXbd, 0);
```

The first two arguments to this routine (**xbdEventCategory** and **xbdEventInstantiated**) are global variables defined in the XBD library that correspond to this event being associated with XBD and triggered when the XBD is instantiated. The third argument (**myDeviceXbdReadyHandler**) is a pointer to the driver routine that is called when this event occurs. The fourth argument (**pMyXbd**) is the parameter that is passed to the routine. In this case, **pMyXbd** is a pointer to the driver-specific device structure. The fifth argument is for option flags, and can normally be left as 0.

In most cases, the routine pointed to by **myDeviceXbdReadyHandler** simply needs to unregister itself from the ERF—using **erfHandlerUnregister( )**—to avoid being triggered again and then unblock the device creation routine. In this way, the device creation routine does not exit until the XBD initialization, which may occur in a different task context, is complete.

#### Advertisement of XBD Methods

The **xbd_funcs** structure is advertised using a call to **xbdAttach( )**:

```
int xbdAttach
    (
    XBD *               xbd,
    struct xbd_funcs *  funcs,
    const char *        name,
    unsigned            blocksize,
    sector_t            nblocks,
    device_t *          result
    )
```

The **xbd** parameter is a pointer to an XBD structure that can be allocated as part of a larger structure describing the device being created. The **funcs** parameter is a pointer to the **xbd_funcs** structure described previously. The next three parameters are self-explanatory. The last parameter, **result**, is a handle for the device that is used, by ERF routines in particular, to identify the device. This parameter is filled in by the **xbdAttach( )** routine.

### xbd_funcs Structure

The XBD library expects that drivers supply a set of function pointers to provide the XBD library with access to devices. These methods are specified in **xbd.h** in the **xbd_funcs** structure as follows:

```
struct xbd_funcs
    {
    int (*xf_ioctl) (struct xbd * dev, int cmd, void * arg);
    int (*xf_strategy) (struct xbd * dev, struct bio * bio);
    int (*xf_dump) (struct xbd * dev, sector_t pos, void * data, size_t
                    size);
    };
```

#### (*xf_ioctl)( )

The **(*xf_ioctl)( )** routine provides a single interface to various driver functions such as device eject, power management, and diagnostic reporting. Much of this functionality is optional and may not apply in all cases. The **ioctl( )** codes used by XBD are defined in **xbd.h**.

#### (*xf_strategy)( )

The **(*xf_strategy)( )** routine provides a way to queue work to the storage driver. As such, you only need to be concerned with managing linked lists containing the queued work for each device under control of your driver. The work to be done is contained in a **bio** structure that is defined in **bio.h**.

#### (*xf_dump)( )

The **(*xf_dump)( )** routine allows the driver to provide a method for writing data to a device in the event of a catastrophic system failure. The underlying routines in your driver must not use any OS services or rely on any interrupt handling.

### ERF New Device Notification

The upper layers of software need to be notified about device creation in the system. This is done by raising an event using the ERF routine **erfEventRaise( )**. For example:

```
erfEventRaise(xbdEventCategory, xbdEventPrimaryInsert, ERF_ASYNC_PROC,
              (void *) pDevice, NULL);
```

The first two arguments (**xbdEventCategory** and **xbdEventPrimaryInsert**) are similar to the first two arguments in **erfHandlerRegister( )**.

The third argument contains a flag indicating what kind of processing is to be performed. The value **ERF_ASYNC_PROC** indicates that this event can be handled asynchronously, and should be used in most cases. If synchronous handling is required, you must specify **ERF_SYNC_PROC**.

The fourth argument (**pDevice**) is the pointer returned in the call to **xbdAttach( )**. The fifth argument is for providing a routine to free the memory pointed to by the fourth argument. Because, in this case, you do not want to free the device pointer, this argument is left as **NULL**.

### 10.8.2 Processing

In addition to providing the **(*xf_strategy)( )** routine, which the XBD layer uses to queue writes and reads to a storage device, the storage class driver must also contain code to process this queued work. In current implementations, this consists of a task that is awakened through a semaphore given at the end of the **(*xf_strategy)( )** routine. This task traverses the linked list of **bio** structures (which contain details on the access to be performed), and executes each one sequentially.

The storage driver typically extracts the sector number, transaction size (in bytes), and transaction direction (read or write) from the **bio** structure. After any necessary checking or conversion of this data (for example, converting the transaction size from bytes to sectors), the driver then calls its low-level read or write routines to complete the transaction.

To mark the **bio** structure as processed, the driver must call the **bio_done( )** routine with a pointer to the **bio** being processed and an **errno** value indicating the result of the processing.

### 10.8.3 Event Reporting

The ERF provides routines that drivers can use to alert higher-level software that events have occurred on the system devices and that these events may require their attention.

An example of this is when a storage device is inserted into the system. In response to this event, the storage controller typically raises an interrupt. As part of the handling of this interrupt, the storage driver ISR can wake up a monitoring task that calls the driver device creation routine. Near the end of this routine, a call to **erfEventRaise( )** should be made to indicate that a new device has been added to the system and is ready for file system mounting.

Similarly, your driver should also use the ERF when a device is removed from the system. One notable difference is that device removal can originate from the operating system as well as the physical connection to the device. The former case is preferred in systems where data integrity is important, as the operating system delays device removal until all device access has stopped. As the driver developer, you can handle this case in the (***xf_ioctl)( )** routine, by calling the **erfEventRaise( )** routine when handling the **XBD_HARD_EJECT** or **XBD_SOFT_EJECT ioctl( )** routines.

Events are raised by making a call to **erfEventRaise( )** as follows:

```
STATUS erfEventRaise
    (
    UINT16    eventCat,          /* Event Category */
    UINT16    eventType,         /* Event Type */
    int       procType,          /* Processing Type */
    void *    pEventData,        /* Pointer to Event Data */
    erfFreePrototype * pFreeFunc /* Function to free Event Data
                                    when done */
    )
```

The XBD library includes several defined event types for use with the ERF. The following event types are passed as the second argument (**eventType**) to **erfEventRaise( )**:

**xbdEventPrimaryInsert**
This event should be raised near the end of the driver's device creation routine to indicate that a new device has been added to the system.

**xbdEventRemove**
This event should be raised during the driver's device deletion routine to indicate that a device has been removed from the system.

**xbdEventInstantiated**
This event should be raised in response to the **XBD_STACK_COMPLETE** (***xf_ioctl)( )** routine, to acknowledge that the driver is ready for access using the XBD interface.

**xbdEventMediaChanged**
This event should be raised when the driver detects that the device's removable media has been removed or replaced.

## 10.9 Writing New Storage Drivers

There is currently no template for storage drivers. For new PCI-based SATA controller drivers, the Silicon Image driver, **vxbSI31xxStorage.c** can be used as a reference. For most on-board Intel ATA/SATA controllers, **vxbIntelIchStorage.c** can be used without modification.

If you are writing a new driver, one strategy is to start with issuing ATA commands. In most cases, you must write a minimum of two routines to handle commands. This includes:

- Command issue—This routine should take (as parameters) the device to which the command is targeted, the command opcode, the desired sector offset, a pointer to a data buffer, and so forth. The routine should then put this information into the format required by the controller. The routine must then complete the necessary register or descriptor accesses to queue this command on the controller.
- Command result—This routine is called when a command is completed. The result of this command may be to fill in a structure with data returned by the controller. In the course of writing this routine, you may also be required to write an ISR that detects a command complete interrupt and calls this routine, or unblocks a task that calls this routine. In some cases, a polling loop can be used instead.

Once these routines are in place, other routines can be included. For example:

- Identify—This routine issues a command to retrieve the physical attributes of the connected device. Because retrieval of physical attributes may be required in several places throughout the driver (for example, when a new device is connected), you may want to put this code into a dedicated routine.
- Read/Write—This routine or set of routines issue either read or write commands. Current Wind River driver implementations use one routine for both read and write. The routine takes a read/write flag as one of its arguments. This implementation reduces redundant code and fits well with the XBD layer, which stores the transaction direction in the **bio** structure.

Once these routines are implemented, the XBD and ERF interfaces can be added to your driver piece by piece. During this stage of development, some of the routines may need to be altered, and you may need to write the monitoring task if you have not done so already.

Your driver should employ the concept of deferring work to a dedicated task. This method produces several advantages, especially if a task (or set of tasks) is

dedicated to each connected device. In this situation, system throughput can be maximized in multiprocessing configurations. However, one important guideline is to ensure that each dedicated task can access only the data structures that belong to it. If this cannot be achieved, some mutual exclusion is required. You may also need to protect accesses to the registers on the controller.

For information on ISR deferral, see the interrupt handling information in *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*.

Once the XBD and ERF interfaces are in place, your driver should be ready for use with VxWorks file systems.



## 10.10 Debugging

When the file system is bypassed, the complexity of debugging storage drivers is decreased. For this reason, the storage driver class includes public low-level access routines.

For example, the Silicon Image driver provides the following sector read/write routine:

```
STATUS sil31xxSectorRW
    (
    int         ctrl,       /* controller number */
    int         port,       /* port number */
    sector_t    sector,     /* sector from which to start access */
    uint32_t    numSecs,    /* number of sectors to access */
    char        *data,      /* data buffer for read or write */
    BOOL        isRead      /* TRUE for read, FALSE for write */
    );
```

Additionally, the routines for block device creation and deletion should be provided by the driver as public routines to aid in debugging.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# 11

# Timer Drivers


11.1 Introduction 214

11.2 Overview 214

11.3 VxBus Driver Methods 215

11.4 Header Files 218

11.5 BSP Configuration 218

11.6 Available Utility Routines 219

11.7 Initialization 219

11.8 Data Structure Layout 219

11.9 Implementing Driver Service Routines 221

11.10 Integrating a Timer Driver 228

11.11 Debugging 232

11.12 SMP Considerations 233

## 11.1 Introduction

This chapter describes timer drivers. This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 11.2 Overview

Timer drivers are used to provide a functional interface between hardware timers and the various operating system services that make use of them.

In VxWorks, timer drivers are used to provide two distinct types of timing services.

- a periodic interrupt timer
- a timestamp timer

Timestamp drivers are used to allow middleware to quickly read a timestamp value from a device, in order to place a timestamp value on an event that has occurred in the system. Timestamp drivers typically provide a high degree of precision, with resolutions of a microsecond or less.

Periodic interrupt timer drivers are used to deliver periodic interrupts to an attached interrupt service routine (ISR). This type of driver is used as the basic *heartbeat interrupt* source for VxWorks. Because VxWorks requires periodic interrupts to perform its scheduling operations, each VxWorks system must include at least one timer driver that supports the generation of periodic interrupts.

This chapter presents the model for both timestamp and periodic interrupt timer drivers. Support for both timestamp and period interrupts can be provided within a single driver, or a driver can choose to provide just one of the two services. A timer driver advertises the services that its hardware supports to the system. Operating system middleware chooses from among the available timers at run time based upon the advertised capabilities of the timer drivers in the system.

Timing hardware often contains more than one timer within a single device. Your timer driver should normally be written so that a single instance supports all of the timers provided by the device hardware.

## 11.3 VxBus Driver Methods

All timer drivers written in accordance with the VxBus model publish a single driver method, **{vxbTimerFuncGet}( )**.

Within a timer driver, the **{vxbTimerFuncGet}( )** method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{vxbTimerFuncGet}
    (
    VXB_DEVICE_ID                       pInst,
    struct vxbTimerFunctionality **     ppTimerFunc,
    int                                 timerNo
    )
```

The **VXB_DEVICE_ID** parameter describes the specific instance (timer device associated with a driver) within the system. Because a single instance of a timer driver can support more than one timer, a **timerNo** parameter is provided in order to identify the specific timer that is being requested within the instance. If a timer driver supports only a single timer, **timerNo** should be tested, and **ERROR** should be returned for all nonzero values of **timerNo**.

Within **func{vxbTimerFuncGet}( )**, the driver fills in the contents of the **vxbTimerFunctionality** structure to describe the capabilities of the requested timer. The fields of the structure are listed in this section, along with a description of each field's use. Refer to *installDir*/**vxworks-6.***x*/**target/h/vxbTimerLib.h** for the type definition for **vxbTimerFunctionality**, and for the following macro definitions:

**BOOL allocated**
This field holds a value that is maintained by the driver. The default value for "allocated" is **FALSE**. When a timer is allocated (using the **(*timerAllocate)( )** function pointer, see *11.9.1 (*timerAllocate)( )*, p.221), the driver sets this field to **TRUE**. When a timer is released (using the (***timerRelease)( )** function pointer, see *11.9.2 (*timerRelease)( )*, p.222), the driver sets this field to **FALSE**.

**UINT32 clkFrequency**
This field holds a value that describes the frequency (in counts per second) that the timer's hardware counter increments (or decrements) when it is running.

**UINT32 minFrequency**
This fields holds a value that describes the minimum frequency (in interrupts per second) that a periodic interrupt timer can support. For timestamp drivers this field is not used.

**UINT32 maxFrequency**

This field holds a value that describes the maximum frequency (in interrupts per second) that a period interrupt timer can support. For timestamp drivers this field is not used.

**UINT32 features**

This field holds a bit-significant value that describes the capabilities of the timer. This value is constructed by performing a logical OR operation on the appropriate values from the following **#define** values found in **vxbTimerLib.h**:

- **VXB_TIMER_CAN_INTERRUPT**—Set if the timer can generate interrupts.
- **VXB_TIMER_INTERMEDIATE_COUNT**—Set if the timer allows the hardware to read values while the counter is running without introducing any skew into the timing results.
- **VXB_TIMER_SIZE_16**, **VXB_TIMER_SIZE_23**, **VXB_TIMER_SIZE_32**—Set if the timer's counter register is 16, 23, or 32 bits (respectively).
- **VXB_TIMER_SIZE_64**—Set if the timer's counter register is 64 bits. When this value is set, the driver must also ensure that non-null values are provided for the **(*timerEnable64)( )**, **(*timerRolloverGet64)( )**, and **(*timerCountGet64)( )** function pointers (see *11.9 Implementing Driver Service Routines*, p.221).
- **VXB_TIMER_CANNOT_DISABLE**—Set if the underlying hardware timer cannot be disabled.
- **VXB_TIMER_STOP_WHILE_READ**—Set if the underlying hardware timer stops incrementing (or decrementing) while the timer is being read.
- **VXB_TIMER_AUTO_RELOAD**—Set if the underlying hardware timer automatically reloads itself when it reaches its terminal count, rather than requiring software intervention to restart the timer.
- **VXB_TIMER_CANNOT_MODIFY_ROLLOVER**—Set if the underlying hardware timer's rollover value is fixed at a single value. This is true for timers that (for example) always count from 0 to their maximum value, rather than to a software-controllable intermediate value.
- **VXB_TIMER_CANNOT_SUPPORT_ALL_FREQS**—Set if the underlying hardware timer cannot be configured to support interrupt frequencies continuously between **minFrequency** and **maxFrequency**.

**UINT32 ticksPerSecond**

This field holds a value that describes the current configuration of the timer hardware, in terms of interrupts per second that the hardware will generate. A timer driver sets this value to a reasonable default (typically between 60 and 100), and maintains the value whenever the requested interrupt delivery rate changes. See the **(*timerEnable)( )** function pointer (see *11.9.6 (*timerEnable)( )*, p.224) for further information.

**char timerName[20]**

This field holds the name of the timer driver. It is used for debug purposes. Timer drivers that support more than one timer within a single driver can choose to create a name that combines the name for the driver with the timer number. Timer drivers that only support a single timer should set this field to the name of the driver.

**UINT32 rolloverPeriod**

This field holds a **UINT32** that describes how long the timer takes to roll over, in seconds. Timers that count quickly have a shorter rollover period than those that count more slowly.



The following function pointers are described in *11.9 Implementing Driver Service Routines*, p.221:

- **STATUS (*timerAllocate)**
- **STATUS (*timerRelease)**
- **STATUS (*timerRolloverGet)**
- **STATUS (*timerCountGet)**
- **STATUS (*timerDisable)**
- **STATUS (*timerEnable)**
- **STATUS (*timerISRSet)**
- **STATUS (*timerEnable64)**
- **STATUS (*timerRolloverGet64)**
- **STATUS (*timerCountGet64)**

After the driver's **func{vxbTimerFuncGet}( )** routine is called, the various supported timer devices associated with the driver are available for allocation.

## 11.4 Header Files

All timer drivers written in accordance with the VxBus model need to include a single header file to provide the data types required for the driver:

```
#include <vxbTimerLib.h>
```

## 11.5 BSP Configuration

Timer drivers do not typically require configuration information from a BSP that is above and beyond the normal device-specific information provided for all drivers. However, when writing a device driver, you should adhere to the existing standard when choosing resource names. The following resource names are used frequently within the existing set of Wind River timer drivers. If your driver allows any of the properties described for these resources to be configured using a BSP resource file, the following strings should be used to query those resources:

**cpuClkRate**
The frequency of the CPU clock, in ticks per second. This is useful in timer drivers where the timing hardware runs at a rate that is correlated with the CPU clock.

**clkRateMin**
The minimum number of interrupts per second that the timer driver hardware can be configured for.

**clkRateMax**
The maximum number of interrupts per second that the timer driver hardware can be configured for.

**clkFreq**
The frequency of the hardware timer.

## 11.6 Available Utility Routines

There are no class-specific utility routines required or available for timer drivers.

## 11.7 Initialization

Timer drivers perform their initialization during the first two phases of VxBus initialization:

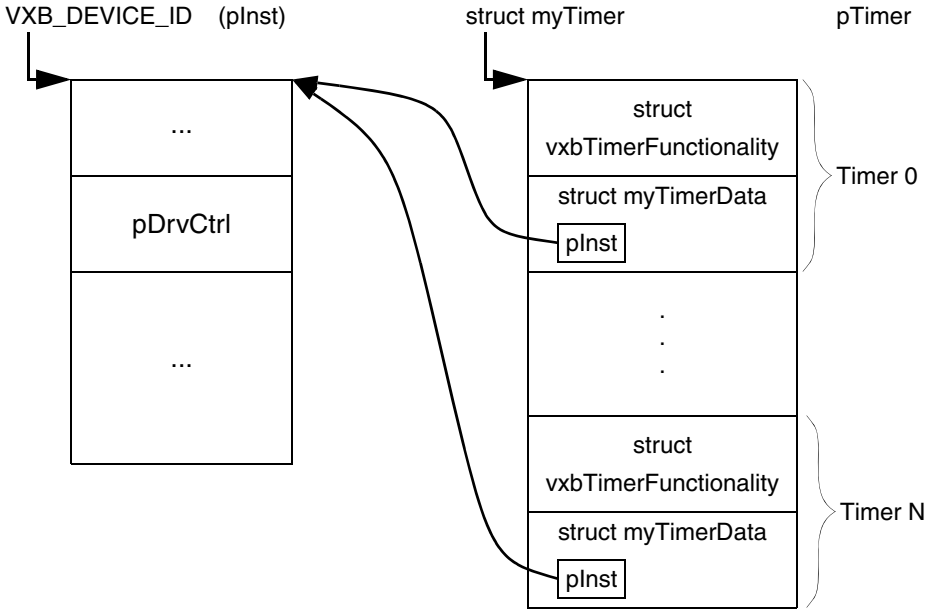
- During VxBus initialization phase 1 (**devInstanceInit( )**), timer drivers should initialize all of their internal data structures, and perform any required initialization of the timer hardware.
- During VxBus initialization phase 2 (**devInstanceInit2( )**), timer drivers should connect their driver-specific ISR to the timer interrupt source(s).

Because any periodic interrupt timer driver can potentially be used as the heartbeat interrupt for the VxWorks kernel, the timer driver must be fully configured by the end of initialization phase 2.

## 11.8 Data Structure Layout

Figure 11-1 describes a recommended layout for the time driver data structure.

Figure 11-1 **Recommended Timer Driver Data Structure Layout**




The two principal elements to this data structure are the **VXB_DEVICE_ID** instance data (by convention referred to as **pInst**), and the driver-specific data structure, which in this figure is labeled as **struct myTimerData**. Note that each of these data structures contains a pointer to the other; the **pInst->pDrvCtrl** field contains a pointer to the driver-specific data structure, and the driver-specific data structure contains a pointer back to the **pInst**.

When a timer driver initializes itself, it typically allocates its **struct myTimer** using **hwMemAlloc( )**, and then initializes the various data structures contained within it. This includes initializing the pointer(s) to **pInst**. Because the **VXB_DEVICE_ID pInst** pointer is not provided to the service routines, the stored pointer(s) to **pInst** is useful to the timer driver service routines.

This documentation in this chapter assumes that the service routines are capable of accessing data within **VXB_DEVICE_ID**, even when **VXB_DEVICE_ID** is not provided as a passed-in parameter to the service routine.

## 11.9 Implementing Driver Service Routines

Once a driver is registered with VxBus and a call is made to the driver's **func{vxbTimerFuncGet}( )** routine, all subsequent interaction between the system and the driver occurs through the routines whose pointers are returned within the **vxbTimerFunctionality** data structure. In the following sections, each of the service routines that can be supported by a timer driver are described. Not all of the service routines need to be implemented in a single driver. For each service routine, a note describing whether the service routine is required for a periodic interrupt driver, a timestamp driver, or for both types of drivers is provided.

### 11.9.1 (*timerAllocate)( )

The **(*timerAllocate)( )** routine is used to allocate a specific timer within a running VxWorks system. Both periodic interrupt and timestamp drivers are required to support this routine.



The prototype for this routine is:

```
STATUS (*timerAllocate)
    (
    VXB_DEVICE_ID   pInst,   /* IN */
    UINT32          flags,   /* IN */
    void **         pCookie, /* OUT */
    UINT32          timerNo  /* IN */
    );
```

This routine tests its input parameters to ensure that it can comply with the requested allocation. If the requested timer (specified by **timerNo**) is available, and if the requested timer supports the requested services (specified by the **flags** parameter), the driver:

- Marks the driver as allocated by setting the **allocated** field to **TRUE** within the **vxbTimerFunctionality** field associated with the timer hardware.
- Configures the timer hardware (if required, based on the **flags** parameter).
- Sets ***pCookie** to the base address of the per-timer data area.
- Returns **OK**.

If the requested timer does not exist, or if the timer cannot be configured according to the properties described in the **flags** parameter, the driver returns **ERROR**.

### 11.9.2 (*timerRelease)( )

The **(*timerRelease)( )** routine is used to release a specific timer that was previously allocated using **(*timerAllocate)( )**. Both period interrupt and timestamp drivers are required to support this routine.

The prototype for this routine is:

```
STATUS (*timerRelease)
    (
    VXB_DEVICE_ID   pInst,   /* IN */
    void *          pCookie  /* IN */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver verifies that the requested timer is allocated. If the timer is allocated, the driver:

- Clears the allocation of the driver by setting the **allocated** field to **FALSE** within the **vxbTimerFunctionality** field associated with the timer hardware.
- Disables delivery of any interrupts from this timer source.
- Clears any associated ISR information from the per-timer data area.
- Returns **OK**.

If the requested timer is not currently allocated, the driver returns **ERROR**.

### 11.9.3 (*timerRolloverGet)( )

The **(*timerRolloverGet)( )** routine is used to query the maximum value that the timer is capable of returning using its **(*timerCountGet)( )** routine (see *11.9.4 (*timerCountGet)( )*, p.223). Timestamp drivers are required to support this routine. This routine is not used for periodic interrupt drivers.

The prototype for this routine is:

```
STATUS (*timerRolloverGet)
    (
    void *   pCookie,        /* IN */
    UINT32 * pCount          /* OUT */
    );
```

The **void *** parameter provided through **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test

both **pCookie** and **pCount** to ensure that they are both non-null. If both pointers are valid, the driver:

- Sets ***pCount** to the maximum value returnable from **(*timerCountGet)( )**.
- Returns **OK**

If either pointer is **NULL**, the driver returns **ERROR**.

### 11.9.4 (*timerCountGet)( )

The **(*timerCountGet)( )** routine is used to query the current value of the timer. Timestamp drivers are required to support this routine. This routine is not used for periodic interrupt drivers.

**NOTE:** In VxWorks, timers always count towards higher numeric values. If the underlying hardware on which the timer is based counts downward, the driver must perform the appropriate mathematics to ensure that the counter appears to count towards higher values from the caller's perspective.



The prototype for this routine is:

```
STATUS (*timerCountGet)
    (
    void *          pCookie,        /* IN */
    UINT32 *        pCount          /* OUT */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. Your driver should test both **pCookie** and **pCount** to ensure that they are both non-null. If both pointers are valid, the driver:

- Sets ***pCount** to the current value of the hardware counter, with appropriate math operations to ensure that counter appears to be counting towards higher values.
- Returns **OK**.

If either **pCookie** or **pCount** are **NULL**, the driver returns **ERROR**.

**NOTE:** Because this routine is used to create timestamps for events that can occur at a high frequency, it should be implemented as efficiently as possible in order to minimize its effect on overall system performance.

When a timer driver is used as the timestamp source for Wind River System Viewer, the kernel makes calls to **(*timerCountGet)( )** at unpredictable times, such as when the kernel has interrupts locked, or while a spinlock is held. To allow **(*timerCountGet)( )** to function correctly when used in this situation, the use of spinlocks is not allowed within **(*timerCountGet)( )**. In addition, the body of **(*timerCountGet)( )** must not perform any operations that result in a Wind River System Viewer event, because this causes an infinite recursion between System Viewer and the timer driver.

For a discussion of event logging and examples of operating system facilities that generate System Viewer events, see the *Wind River System Viewer User's Guide*.

### 11.9.5 (*timerDisable)( )

The **(*timerDisable)( )** routine is used to disable interrupts generated by the underlying timer hardware. Periodic interrupt drivers are required to support this routine. This routine is not used for timestamp drivers.

The prototype for this routine is:

```
STATUS (*timerDisable)
    (
    void *  pCookie         /* IN */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test **pCookie** to ensure that it is non-null. If **pCookie** is valid, the driver:

- Disables interrupt generation for the requested hardware timer.
- Returns **OK**.

If **pCookie** is **NULL**, the driver returns **ERROR**.

### 11.9.6 (*timerEnable)( )

The **(*timerEnable)( )** routine is used to enable generation of interrupts by the underlying timer hardware. Periodic interrupt drivers are required to support this routine. This routine is not used for timestamp drivers.

The prototype for this routine is:

```
STATUS (*timerEnable)
    (
    void *  pCookie,        /* IN */
    UINT32  maxTimerCount   /* IN */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test **pCookie** to ensure that it is non-null. If **pCookie** is valid, the driver:

- Programs the underlying timer hardware so that it generates an interrupt each time **maxTimerCount** counts have occurred within the timer.
- Enables interrupt generation for the requested hardware timer.
- Returns **OK**.

If **pCookie** is **NULL**, the driver returns **ERROR**.



### 11.9.7 (*timerISRSet)( )

The **(*timerISRSet)( )** routine is used to connect an ISR to the underlying timer hardware interrupt. Both periodic interrupt and timestamp drivers are required to support this routine (if your hardware supports interrupt generation).

The prototype for this routine is:

```
STATUS (*timerISRSet)
    (
    void *  pCookie,        /* IN */
    void   (*pIsr)(int),    /* IN */
    int     arg             /* IN */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The **pIsr** and **arg** parameters are caller-provided values that should be stored within the per-timer data area so that the values can be retrieved during interrupt handling for the timer hardware.

After **(*timerISRSet)( )** is called to connect the requested ISR to the timer interrupt, the specified ISR is called each time a timer interrupt occurs, using the following code fragment:

```
(*pIsr)(arg);
```

### 11.9.8 (*timerEnable64)( )

The **(*timerEnable64)( )** routine is used to enable generation of interrupts by the underlying timer hardware for timers that support 64-bit counters. Support for 64-bit timers is optional. As such, neither timestamp drivers nor periodic interrupt drivers are required to support this routine.

If you want to include support for 64-bit timers, this routine should be supported by the driver. In addition, the **VXB_TIMER_SIZE_64** property should be added to the **features** field of the **vxbTimerFunctionality** structure that is returned from **func{vxbTimerFuncGet}( )**.

The prototype for this routine is:

```
STATUS (*timerEnable64)
    (
     void *  pCookie,        /* IN */
     UINT64  maxTimerCount   /* IN */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test **pCookie** to ensure that it is non-null. If **pCookie** is valid, the driver:

- Programs the underlying timer hardware so that it generates an interrupt each time **maxTimerCount** counts occur within the timer.
- Enables interrupt generation for the requested hardware timer.
- Returns **OK**.

If **pCookie** is **NULL**, the driver returns **ERROR**.

### 11.9.9 (*timerRolloverGet64)( )

The **(*timerRolloverGet64)( )** routine is used to query the maximum value that the timer is capable of returning using its **(*timerCountGet64)( )** routine. Support for 64-bit timers is optional. As such, neither timestamp drivers nor periodic interrupt drivers are required to support this routine.

If you want to include support for 64-bit timers, this routine should be supported by the driver. In addition, the **VXB_TIMER_SIZE_64** property should be added to the **features** field of the **vxbTimerFunctionality** structure that is returned from **func{vxbTimerFuncGet}( )**.

The prototype for this routine is:

```
STATUS (*timerRolloverGet64)
    (
    void *   pCookie,        /* IN */
    UINT64 * pCount          /* OUT */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test both **pCookie** and **pCount** to ensure that they are both non-null. If both pointers are valid, the driver:

- Sets ***pCount** to the maximum value returnable from **(*timerCountGet64)( )**.
- Returns **OK**

If either pointer is **NULL**, the driver returns **ERROR**.



### 11.9.10 (*timerCountGet64)( )

The **(*timerCountGet64)( )** routine is used to query the current value of the timer for 64-bit timers. Support for 64-bit timers is optional. As such, neither timestamp drivers nor periodic interrupt drivers are required to support this routine.

**NOTE:** In VxWorks, timers always count towards higher numeric values. If the underlying hardware on which the timer is based counts downward, the driver must perform the appropriate mathematics to ensure that the counter appears to count towards higher values from the caller's perspective.

The prototype for this routine is:

```
STATUS (*timerCountGet64)
    (
    void *          pCookie,        /* IN */
    UINT64 *        pCount          /* OUT */
    );
```

The **void *** parameter provided using **pCookie** points to the per-timer data area previously returned through a call to **(*timerAllocate)( )**. The driver should test both **pCookie** and **pCount** to ensure that they are both non-null. If both pointers are valid, the driver:

- Sets ***pCount** to the current value of the hardware counter, with appropriate math operations to ensure that counter appears to be counting towards higher values.
- Returns **OK**.

If either **pCookie** or **pCount** are **NULL**, the driver returns **ERROR**.

**NOTE:** Because this routine is used to create timestamps for events that can happen at a high frequency, it should be implemented as efficiently as possible in order to minimize its effect on overall system performance.

## 11.10 Integrating a Timer Driver

Traditionally, VxWorks uses between one and three different timers in a running system. All VxWorks operating systems use a standard periodic interrupt timer driver to support the kernel's heartbeat interrupt. Additionally, if a system is configured to support an auxiliary timer or timestamp driver, these services also make use of the timer drivers that are implemented according to this chapter. The following sections discuss the integration of timer drivers to the system clock, auxiliary clock, and to the timestamp driver.

### 11.10.1 VxWorks System Clock

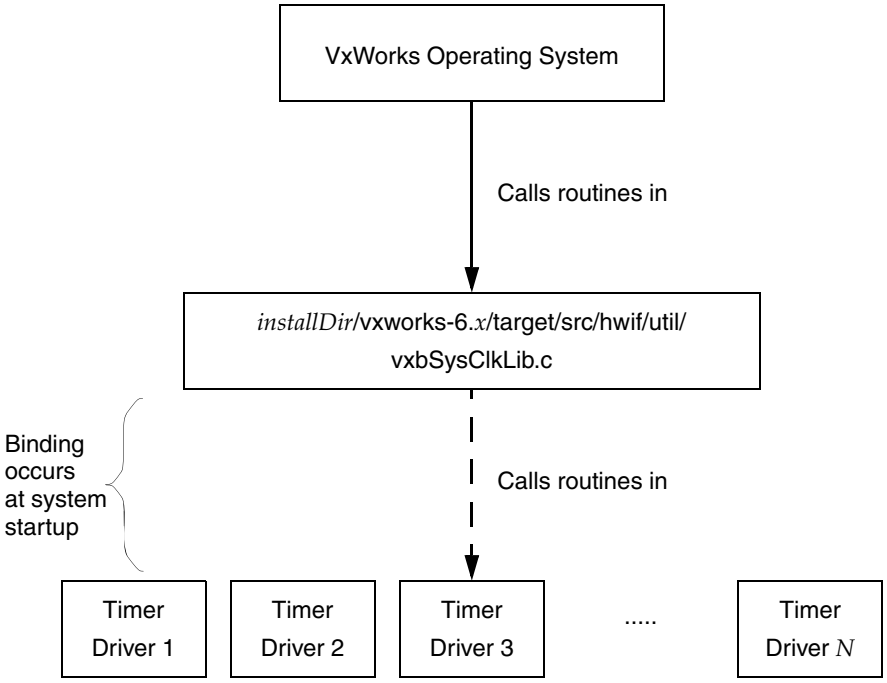
Prior to VxWorks 6.5, the system clock is commonly implemented directly within the BSP. The BSP is expected to either directly implement (or to include using a **#include**) the following set of system clock (**sysClk*( )**) routines:

- **sysClkConnect( )**
- **sysClkEnable( )**
- **sysClkDisable( )**
- **sysClkRateSet( )**
- **sysClkRateGet( )**

As of VxWorks 6.6, this model has been enhanced to allow the kernel's system clock to be implemented within the BSP, or by using a VxBus timer driver as described in this chapter. In this release, one implementation of the system clock API is implemented within *installDir*/**vxworks-6.***x*/**target/src/hwif/util/vxbSysClkLib.c**. All of the system clock routines listed previously are implemented in **vxbSysClkLib.c**. In addition to the required system clock routines, this library contains code to allocate a periodic interrupt timer driver during system startup, and to connect this timer driver to the system clock routines.

Graphically, the system clock consists of three layers as shown in Figure 11-2.

Figure 11-2 **VxWorks System Clock Hierarchy**




During system initialization, the available periodic interrupt timer sources are scanned by the system clock library, and one of the available timer sources is selected for use as the timer source for the heartbeat interrupt.

During driver development, you may wish to force the system clock library to choose your periodic interrupt timer driver, rather than one of the other available timer drivers in the system. The system clock library supports this feature through the use of three global variables that are defined within the library as follows:

```
char *  pSysClkName = NULL;
UINT32  sysClkDevUnitNo = 0;
UINT32  sysClkTimerNo  = 0;
```

If these global variables are redefined in a BSP during execution of the **sysHwInit( )** routine, the **vxbSysClkLib** library uses the timer driver that is described by the global variables instead of the one found through its matching algorithm. The **vxbSysClkLib** library performs a case-sensitive string comparison

of **pSysClkName** with the names of each of the available timer drivers (as described by the **timerName** field in the **vxbTimerFunctionality** structure returned by the driver). If the driver name matches the name specified using **pSysClkName**, the **vxbSysClkLib** library compares the unit number and the timer number of the driver against the values specified by BSP. If an exact match is found, the **vxbSysClkLib** library uses the specified timer.

Note that if an exact match is not found, **vxbSysClkLib** reverts to using its matching algorithm, rather than failing to connect the kernel's system clock to an underlying timing source. If this occurs, the **vxbSysClkLib** library post an error detection and reporting message indicating the failure to find the requested timer.

### 11.10.2 VxWorks Auxiliary Clock

Prior to VxWorks 6.5, the auxiliary clock is commonly implemented directly within the BSP. The BSP is expected to either directly implement (or to include using a **#include**) the following set of **sysAuxClk*( )** routines:

- **sysAuxClkConnect( )**
- **sysAuxClkEnable( )**
- **sysAuxClkDisable( )**
- **sysAuxClkRateGet( )**
- **sysAuxClkRateSet( )**

As of VxWorks 6.6, this model has been changed to allow the kernel's auxiliary clock to be implemented using a VxBus timer driver as described in this chapter. In this release, the VxWorks auxiliary clock API is implemented within *installDir*/**vxworks-6.***x*/**target/src/hwif/util/vxbAuxClkLib.c**. All of the **sysAuxClk*( )** routines listed previously are implemented in **vxbAuxClkLib.c**. In addition to the required **sysAuxClk*( )** routines, this library contains code to allocate a periodic interrupt timer driver during system startup, and to connect this timer driver to the **sysAuxClk*( )** routines.

During system initialization, the available periodic interrupt timer sources are scanned by the **vxbAuxClkLib** library, and one of the available timer sources is selected for use as the timer source for the auxiliary clock.

During driver development, you may wish to force the **vxbAuxClkLib** library to choose your periodic interrupt timer driver, rather than one of the other available timer drivers in the system. The **vxbAuxClkLib** library supports this feature

through the use of three global variables that are defined within the library as follows:

```
char *  pAuxClkName = NULL;
UINT32  auxClkDevUnitNo = 0;
UINT32  auxClkTimerNo  = 0;
```

If these global variables are redefined in a BSP during execution of the BSP **sysHwInit( )** routine, the **vxbAuxClkLib** library uses the timer driver that is described by the global variables instead of the one found through its matching algorithm. The **vxbAuxClkLib** library performs a case-sensitive string comparison of **pAuxClkName** with the names of each of the available timer drivers (as described by the **timerName** field of the **vxbTimerFunctionality** returned by the driver). If the driver name matches the name specified using **pAuxClkName**, the **vxbAuxClkLib** library compares the unit number and the timer number of the driver against the values specified by the BSP. If an exact match is found, the **vxbAuxClkLib** library uses the specified timer.

Note that if no exact match is found, the **vxbAuxClkLib** library reverts to using its matching algorithm, rather than failing to connect the kernel's system clock to an underlying timing source. If this occurs, the **vxbAuxClkLib** library posts an error detection and reporting message indicating the failure to find the requested timer.



### 11.10.3 VxWorks Timestamp Driver

Prior to VxWorks 6.5, the system timestamp driver is commonly implemented directly within the BSP. The BSP is expected to either directly implement (or to include using a **#include**) the following set of timestamp routines:

- **sysTimestampConnect( )**
- **sysTimestampEnable( )**
- **sysTimestampDisable( )**
- **sysTimestampPeriod( )**
- **sysTimestampFreq( )**
- **sysTimestamp( )**
- **sysTimestampLock( )**

As of VxWorks 6.6, this model has been changed to allow the system timestamp to be implemented using a VxBus timer driver as described in this chapter. In this release, the VxWorks timestamp driver API is implemented in *installDir*/**vxworks-6.***x*/**target/src/hwif/util/vxbTimestampLib.c**. All of the routines listed previously are implemented in **vxbTimestampLib.c**. In addition to the required **sysTimestamp*( )** routines, this library contains code to allocate a

timestamp timer driver during system startup, and to connect this timer driver to the **sysTimestamp*( )** routines.

During system initialization, the available timestamp timer sources are scanned by the **vxbTimestampLib** library, and one of the available timer sources is selected for use as the timer source for the timestamp.

During driver development, you may wish to force the **vxbTimestampLib** library to choose your timestamp timer driver, rather than one of the other available timer drivers in the system. The **vxbTimestampLib** library supports this feature through the use of three global variables that are defined within the library as follows:

```
char *  pTimestampTimerName = NULL;
UINT32  timestampDevUnitNo = 0;
UINT32  timestampTimerNo = 0;
```

If these global variables are redefined in a BSP during execution of the BSP **sysHwInit( )** routine, the **vxbTimestampLib** library uses the timer driver that is described by the global variables instead of the one found through its matching algorithm. The **vxbTimestampLib** library performs a case-sensitive string comparison of **pSysClkName** with the names of each of the available timer drivers (as described by the **timerName** field in the **vxbTimerFunctionality** structure returned by the driver). If the driver name matches the name specified using **pSysClkName**, the **vxbTimestampLib** library compares the unit number and the timer number of the driver against the values specified by BSP. If an exact match is found, the **vxbTimestampLib** library uses the specified timer.

Note that if no exact match is found, the **vxbTimestampLib** library reverts to its matching algorithm, rather than failing to connect the timestamp driver to an underlying timing source. If this occurs, the **vxbTimestampLib** library posts an error detection and reporting message indicating the failure to find the requested timer.

## 11.11 Debugging

When debugging a timer driver, as with all driver development, it is convenient to have a fully functional system to test the driver on. A fully functioning system provides you with full access to the debug capabilities of VxWorks as well as the VxBus show routines. Because timer drivers can be allocated to the VxWorks system clock during system boot, use a functional timer driver as the VxWorks

system clock, so that VxWorks can boot into a fully functional system that you can then use to debug your timer driver.

The simplest way to prevent your driver from being selected as the system clock is to delay the registration of your timer driver until after the system has booted. When you delay registration, your driver is unavailable during the period when the system clock is evaluated, therefore it cannot be selected as the system clock.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

## 11.12 SMP Considerations



In VxWorks SMP, any CPU in the system can utilize the services of a timestamp driver. This can present a unique problem if CPU-specific registers are used to implement a timestamp service. For example, on the MIPS architecture, the CPU **C0_COUNT** and **C0_COMPARE** registers are often used for timestamping. However, these registers are not necessarily synchronized across the various cores in an SMP system. If these registers are not synchronized in the SMP system, the timestamp driver using these registers returns inconsistent results unless it is only used on a single CPU within the system.

If CPU-specific registers are used as a time base for a timer driver, the registers must be synchronized across all CPUs in the SMP system. The steps required to synchronize these registers is, by definition, architecture and CPU-specific. Unless your driver handles this situation, you should not advertise the **VXB_TIMER_INTERMEDIATE_COUNT** capability when compiled for SMP.

For more information on SMP considerations for drivers, see *VxWorks Device Driver Developer's Guide: Device Driver Fundamentals*. For more information on the optional VxWorks SMP product as a whole, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

# 12 USB Drivers

12.1 Introduction 235
12.2 Wind River USB Overview 236
12.3 Host Controller and Root Hub Class Drivers 237

## 12.1 Introduction

This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

This chapter includes general information on the Wind River USB product with respect to device driver development. The focus of the chapter is on those USB device drivers that comply with the VxBus device driver model. Other driver types are mentioned briefly. For complete information on Wind River USB, including information on device drivers that do not conform to the VxBus driver model, see the *Wind River USB Programmer's Guide*.

## 12.2 Wind River USB Overview

Wind River USB provides support for the Universal Serial Bus for both USB transaction initiators (USB hosts) and to allow a VxWorks target to act as a USB peripheral. The USB host (sometimes called the USB host stack) and the USB peripheral (sometimes called the USB peripheral stack) conform to the USB 2.0 specification and, depending on the hardware, offer data rates up to 480 MB/s.

**NOTE:** The USB host and the USB peripheral code do not support the USB On-The-Go (OTG) HNP or SRP protocols.

### 12.2.1 USB Host Stack Drivers

The USB host stack consists of the USB driver (USBD), host controller drivers, hub drivers, and class drivers.

Wind River provides host controller drivers (HCDs) for the Enhanced Host Controller Interface (EHCI), the Universal Host Controller Interface (UHCI), and the Open Host Controller Interface (OHCI). In addition, Wind River USB provides root hub drivers for the USB controllers, a generic hub class driver, and a collection of class drivers for various types of USB peripherals.

#### VxBus Model Drivers

USB host controller drivers and root hub drivers comply with the VxBus device driver model. These drivers are discussed further in, see *12.3 Host Controller and Root Hub Class Drivers*, p.237.

#### Other Host Drivers

USB class drivers do not adhere to the VxBus model. Instead, they rely on the interfaces associated with the USBD to connect USB devices to the appropriate class driver. For more information on writing USB class drivers, see the *Wind River USB Programmer's Guide*.

The USBD is not a true device driver in the sense that it does not directly control hardware. Rather it serves as the central interface layer between the various USB components. There is one and only one USBD in each VxWorks USB host image.

The USBD is not a VxBus model driver. The driver is started in a manner similar to an application program.

**NOTE:** For information on the USBD, consult the USB Specification 2.0 available from **http://www.usb.org**.

### 12.2.2 USB Peripheral Stack Drivers

The Wind River target or peripheral stack provides drivers for a number of target controller drivers (TCDs) as well as emulation software for a variety of target devices such as bulk storage, printers, and so forth. TCDs are not VxBus compliant and are not discussed in this documentation. For information on writing target controller drivers, see the *Wind River USB Programmer's Guide*.



## 12.3 Host Controller and Root Hub Class Drivers

USB devices are initially plugged into a USB hub or root hub. The USB host reads the configuration descriptors, device descriptors, and interface descriptors from the device and attaches the device to the appropriate class driver. The class driver then issues commands to the USBD which in turn commands the appropriate host controller driver and hub driver to perform the USB transactions necessary for the system to use the device.

USB host controller drivers are VxBus-compliant and are used by the USBD to execute USB transactions in conjunction with the hub drivers.

Root hub drivers are also VxBus compliant. The root hub drivers are subordinate to the host controller driver and are loaded by VxBus during the instantiation of the host controller.

Wind River provides a generic hub class driver that is instantiated as needed to support downstream hubs. Both the host controller drivers and the hub drivers are controlled by the USBD through calls to and from the class drivers. The USBD interface software provides an API that is used to interface with both the host controller drivers and the hub. Application code is not typically aware of the controller or hub on which the transactions take place.

### 12.3.1 VxBus Driver Methods

The host controller drivers and the root hub drivers define the **{vxbDrvUnlink}( )** driver method. This routine is responsible for the orderly shutdown of the host controller hardware and de-registration of the bus with the USBD.

In the case of the VxBus root hub class driver, **func{vxbDrvUnlink}( )** is responsible for disconnecting all downstream devices as well as the hub itself. This routine is called when a hub is disconnected from USB. The **func{vxbDrvUnlink}( )** routine performs the disconnect by calling the remove routine provided by each class driver for each device connected to the hub.

```
VOID func{vxbDrvUnlink}
    (
    VXB_DEVICE_ID devID
    )
```

### 12.3.2 Header Files

The host controller drivers (HCDs) and root hub class drivers use an operating system abstraction layer (OSAL) that provides customized operating system services to the drivers. These services are defined in **usbOsal.h**.

```
#include <usb/usbOsal.h>
```

**NOTE:** Care should be taken to use the operating system services provided in the OSAL rather than the corresponding VxWorks services. In particular, **OS_MALLOC( )** should be used rather than the general purpose **malloc( )** routine. Failure to use these services correctly can result in unpredictable system behavior.

The HCD and root hub class drivers are responsible for both initiating action on and executing commands issued by the USBD. The USBD interface is defined in **usbHst.h**.

```
#include <usb/usbHst.h>
```

### 12.3.3 BSP Configuration

The majority of USB HCDs reside on the PCI bus. For these devices, the VxBus integration with the USB HCDs takes care of the HCD registration, host controller device detection, base address mapping, interrupt connection, and so forth. This leaves the BSP developer to write the address translation routines that translate CPU addresses to addresses used by the HCD and vice versa, if necessary.

**NOTE:** Prior to VxBus implementation, these routines were contained in the BSP-specific **usbPciStub.c** file and were called **usbMemToPci( )** and **usbPciToMem( )**. In non-PCI bus versions, these routines were called **usbMemToBus( )** and **usbBusToMem( )**.

Should a mapping be necessary, the default translation methods can be overridden in the BSP **hwconf.c** file with entries as shown in the following example. The resource **cpuToBus** translates a CPU address to an address understandable by the HCD. The resource **busToCpu** translates an HCD address to an address understandable by the CPU.

```
const struct hcfResource pentiumPci0Resources[] =
{
    …
    …
#ifdef INCLUDE_USB
    { "cpuToBus", HCF_RES_ADDR, {(void *) usbMemToPci}},
    { "busToCpu", HCF_RES_ADDR, {(void *) usbPciToMem}},
#endif
}
```



Some USB HCDs reside on the PLB rather than the PCI. In these cases, additional **hwconf.c** entries are needed.

The first of these entries informs VxBus of the existence of the PLB device.

```
const struct hcfDevice hcfDeviceList[] = {
    …
    …
    { "vxbPlbUsbXXXX", 0, VXB_BUSID_PLB, 0, vxbPlbUsbXXXDevNum0,
    vxbPlbUsbXXXDevResouces0},
    { "vxbPlbUsbXXXX", 1, VXB_BUSID_PLB, 0, vxbPlbUsbXXXDevNum1,
    vxbPlbUsbXXXDevResources1},
    …
}
```

The next entries provide the base address, interrupt vector, interrupt level information, and so forth for each of the devices described in **hcfDeviceList[ ]**. If necessary, the address conversion routines are also defined.

```
const struct hcfResource vxbPlbUsbXXXDevResouces0 [] = {
    { "regBase", HCF_RES_INT, { (void *)0x8000 }},
    { "irq", HCF_RES_INT, {(void *)INUM_TO_IVEC(INT_NUM_0)}},
    { "irqLevel", HCF_RES_INT, {(void *)INT_NUM_0}},
    { "cpuToBus", HCF_RES_ADDR, {(void *) usbMemToBus}},
    { "busToCpu", HCF_RES_ADDR, {(void *) usbBusToMem}},
};
#define vxbPlbUsbXXXDevNum0 NELEMENTS(vxbPlbUsbXXXDevResouces0)
```

### Endian Conversion for USB Data Transfers

The data sent by the USB host stack over the PCI bus is always in little-endian format. This behavior conforms to the USB specification. When a big-endian target is used, the data sent to the USB host must be converted from big-endian to little-endian in the BSP. Conversely, any data passed from the USB host to the target must be converted from little-endian to big-endian format. The HCDs handle this conversion. The BSP developer does not need to do anything at the BSP level for endianness conversion in either direction.

**NOTE:** No endianness conversion is required for data transfer over the PLB bus because this bus is directly mapped to the CPU.

#### Prototypes for Address Conversion Routines

Prototypes for the address conversion routines are as follows:

**NOTE:** If the system memory and the bus on which the HCD resides are mapped one-to-one, the BSP developer does not need to provide these routine definitions.

```
/*************************************************************************
*
* usbMemToBus - Convert a memory address to a bus- reachable memory address
*
* Converts <pMem> to an equivalent 32-bit memory address visible from the
* PLB bus. This conversion is necessary to allow PLB bus masters to address
* the same memory viewed by the processor.
*
* RETURNS: converted memory address
*/

UINT32 usbMemToBus
(
    pVOID pMem  /* memory address to convert */
);

/*********************************************************
* usbBusToMem - Convert a PLB address to a CPU-reachable pointer
*
* Converts <plbAdrs> to an equivalent CPU memory address.
*
* RETURNS: pointer to PLB memory
*/

pVOID usbBusToMem
(
    UINT32 plbAdrs  /* 32-bit PLB address to be converted */
);
```

### 12.3.4 Available Utility Routines

The **usbTool** utility provides a mechanism to manipulate the USB HCDs and class drivers. For more information on this tool, see the *Wind River USB Programmer's Guide*.

### 12.3.5 Initialization

Initialization takes place in the following phases:

1. Registration with VxBus.

   The initialization code registers the desired controller with VxBus. This makes VxBus aware that the hardware driver is available. This registration happens during the first phase of VxBus initialization and uses minimal operating system support. At this stage, only the host controller drivers (HCDs) are registered with VxBus. HCD initialization happens later, once complete operating system support is available.

2. Initialization of the USB host controller devices.

   The host controller devices are initialized during the third phase of VxBus initialization. Before this initialization, VxBus:

   - Initializes the USB host stack by calling the **usbInit( )** routine.
   - Initializes the particular host controller driver by calling the driver initialization routine.

3. Initialization of the USB class drivers.

   This happens in VxBus phase 3 initialization. At this stage, the class drivers included in the VxWorks configuration are initialized.

Note that the root hubs are discovered by VxBus during phase 3, once the USBs on the host controller drivers are instantiated. The phase 3 root hub connect routine is responsible for discovering all downstream devices. Because all hubs are subordinate to the USB host controller drivers, the initialization and connection routines are invoked as the host controller drivers are connected.

The USB host controllers are somewhat unique in that phase 2 device initialization relies on data structures and values contained in the USBD. Therefore, the USBD must be started prior to phase 2 initialization of the host controller drivers. During system startup, this call is made automatically as an artifact of the **INCLUDE_USB_INIT** definition.

**NOTE:** USB startup does not need to occur during system boot. It is common to invoke the initialization routines from the VxWorks development shell. Most importantly, the first step—registration with VxBus—can be deferred.

### 12.3.6 Debugging

Including the **INIT** macros in the BSP configuration initializes the USB components at boot time. However, you can also defer the USB stack initialization and initialize the components from the VxWorks development shell when debug utilities are available.

The macros **INCLUDE_UHCI**, **INCLUDE_EHCI**, and **INCLUDE_OHCI** ensure that the corresponding host controller driver initialization is included in the VxWorks image. They do not, however, initialize any part of the component. To initialize a controller after system start, it is necessary to call the USB stack initialization routine, the controller initialization routine, and the VxBus registration routine, in that order.

The following example shows how to use the VxWorks development shell to initialize the USB stack and the EHCI controller.

Initialize the USB stack:

```
-> usbInit
value = 0 = 0x0
```

Initialize the EHCI host controller driver:

```
-> usbEhcdInit
value = 0 = 0x0
```

Register the EHCI driver with VxBus:

```
-> vxbUsbEhciRegister
value = 0 = 0x0
```

List the USB host controllers and devices:

```
-> vxBusShow
Registered Bus Types:
USB-EHCI_Bus @ 0x02698440
USB-HUB_Bus @ 0x00455708
PCI_Bus @ 0x0044f89c
MII_Bus @ 0x0045243c
Local_Bus @ 0x0044f6b0
```

```
Registered Device Drivers:
vxbPciUsbEhci at 0x00455644 on bus PCI_Bus, funcs @ 0x0045559c
vxbPlbUsbEhci at 0x00455604 on bus Local_Bus, funcs @ 0x0045559c
vxbUsbEhciHub at 0x004555c4 on bus USB-EHCI_Bus, funcs @ 0x004555a8
vxbUsbHubClass at 0x0048c198 on bus USB-HUB_Bus, funcs @ 0x004556ec
.
.
.
Busses and Devices Present:
Local_Bus @ 0x00471b70 with bridge @ 0x0044f718
Device Instances:
ns16550 unit 0 on Local_Bus @ 0x00472b30 with busInfo 0x00000000
ns16550 unit 1 on Local_Bus @ 0x00472d30 with busInfo 0x00000000
pentiumPci unit 0 on Local_Bus @ 0x00472f30 with busInfo 0x00471db0
i8253TimerDev unit 0 on Local_Bus @ 0x00476330 with busInfo 0x00000000
fileNvRam unit 0 on Local_Bus @ 0x00476430 with busInfo 0x00000000
.
.
.
USB-EHCI_Bus @ 0x00477470 with bridge @ 0x00473730
Device Instances:
vxbUsbEhciHub unit 1 on USB-EHCI_Bus @ 0x026b24c0 with busInfo 0x004774b0
Orphan Devices:
USB-HUB_Bus @ 0x004774b0 with bridge @ 0x026b24c0
Device Instances:
Orphan Devices:
USB-EHCI_Bus @ 0x00477570 with bridge @ 0x00475b30
Device Instances:
vxbUsbEhciHub unit 1 on USB-EHCI_Bus @ 0x026cca00 with busInfo 0x004775b0
Orphan Devices:
.
.
.
```

Before registration, the USB devices that are present in the system show up under **vxBusShow( )** as orphan devices. After being invoked from the shell, the registration of the device driver causes VxBus to discover those orphan devices and attempt to initialize them. In this situation, the failures that occur during initialization can be examined with the tools available from Workbench and in the VxWorks development shell.

# 13 Other Driver Classes


13.1 Introduction 245

13.2 Overview 246

13.3 VxBus Driver Methods 246

13.4 Header Files 247

13.5 BSP Configuration 247

13.6 Available Utility Routines 248

13.7 Initialization 248

13.8 Debugging 248


## 13.1 Introduction

Earlier chapters in this volume describe the classes of drivers that are already defined for use within the VxBus framework. However, there are other kinds of devices that do not fit well into any of the supported categories. This chapter discusses these drivers, which are referred to as other-class drivers.

This chapter assumes that you are familiar with the contents of the *VxWorks Device Driver Developer's Guide, Volume 1: Fundamentals of Writing Device Drivers*, which discusses generic driver concepts as well as details of VxBus that are not specific to any driver class.

## 13.2 Overview

Other-class drivers include devices such as digital-to-analog converters and analog-to-digital (D/A and A/D) converters, robot control systems, and so on. There are also devices that are completely unique to a given application, such as the rock abrasion tool on the Mars rovers.

When writing a driver for one of these devices, there is no existing framework to indicate how the driver fits in with the rest of the system. This can cause some difficulties while eliminating others when compared to development for supported driver classes.

In many cases, an other-class driver is written to manage a device for a single, specific application, therefore there is no requirement that the driver be written in a portable or cross-platform manner. When this is the case, the application and driver can be designed so that they share a set of APIs, and the driver and application can communicate using those APIs. These APIs typically have no constraints resulting from interactions with other modules.

However, it can also be more difficult to develop these other-class drivers when compared with the predefined classes. This is the case when you want your driver to be more loosely coupled with the application. You may have a situation where multiple drivers of the same type are used, therefore each driver cannot provide the global symbols of the API that it would provide if it were the only driver of the class on a given system. Or you may have a requirement that the driver be available only for high-end configurations.

Requirements such as these place additional constraints on the device driver developer. These constraints must be handled in a manner suitable for the particular application, and are not described in this manual.

## 13.3 VxBus Driver Methods

When developing an other-class driver, you can make use of the **{driverControl}( )** driver method to perform any specific functionality that you choose.

The **func{driverControl}( )** routine provided by your driver takes an argument of a structure pointer, where the structure contains a driver name field, a command field, and a pointer field.

The driver name field must be set to the name of the driver. The command field is an integer description of the functionality that is requested, and is driver specific. The pointer field is defined as type **void ***, and can be cast to any structure type required by the application and driver.

To use VxBus communication mechanisms between your driver and application, your custom driver can advertise the **{driverControl}( )** driver method. The **func{driverControl}( )** routine, when called, checks the functionality name and driver name fields to verify that the structure provides the requested information. If the requested driver name and functionality match those provided by the driver, the driver fills in the fields of the structure with the appropriate data and uses the data in the structure to perform some action.

By using this mechanism, your driver can provide an API of function pointers and identification arguments to those routines, which are kept in a structure. The application gains access to this API by calling **vxbDevIterate( )**, searching for the **{driverControl}( )** method and, if there is a match, calling the **func{driverControl}( )** with the appropriate functionality name and driver name. For an example of how to use this mechanism, see the sample driver in *installDir*/**vxworks-6.***x*/**target/3rdparty/windriver/sample**.



## 13.4 Header Files

There are no header files specific to the other-class driver class.

## 13.5 BSP Configuration

Other-class drivers must conform to the normal BSP configuration constraints for all drivers. For more information on BSP configuration, see *VxWorks Device Driver Developer's Guide (Vol.1): Device Driver Fundamentals*.

## 13.6 Available Utility Routines

There are no utility routines specific to this driver class.

## 13.7 Initialization

The normal initialization sequence applies to other-class drivers. There are no pre-existing restrictions on when each part of the initialization must occur, other than those limiting what external system resources are available, such as the use of semaphores and other kernel services prohibited from phase 1 initialization.

In many cases, when developing other-class drivers you may decide to perform your initialization during VxBus phase 3 initialization. This allows the kernel to be brought up and the application started, while the driver initialization only occurs when the kernel or application initialization is blocked. The you or the application designer must provide some mechanism for the application to know when the driver's services are available.

## 13.8 Debugging

There are no debugging hints specific to this class.

For general driver debugging information, see *VxWorks Device Driver Developer's Guide (Vol. 1): Development Strategies*.

# A Glossary

**access routine**

A routine provided by VxBus that a driver calls in order to access or manipulate a device register.

**advertise**

Make available to VxBus, as with a *driver method*.

**bus**

A hardware mechanism for communication between the processor and a device, or between different devices. This term can also apply to processor-to-processor communication, such as with RapidIO or the processor local bus (PLB) on SMP and AMP systems.

**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. The bus controller driver is also responsible for enumerating devices present on the bus. See also *device*, *driver*, *enumeration*, and *instance*.

**bus discovery**

See *enumeration*.

**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 be paired to form an instance.

**bus type**

A kind of bus, such as PCI or RapidIO. See also *bus controller*.

**child**

A device that is attached to a bus.

**cluster**

Buffers used by **netBufLib** to hold packet data. See also *mBlk*.

**descriptor**

For DMA, a descriptor is a data structure shared by the device and driver, which communicates the size, location, and other characteristics of data buffers used to hold transmit and receive data. The data format is defined by the design of the device.

**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 farther from the CPU on the bus hierarchy. See also *child*.

**driver**

A compiled software module along with the infrastructure required to make the driver visible to Workbench and BSPs. The software module usually includes 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. The infrastructure typically includes a CDF that allows integration with Workbench and **vxprj**, and stub files for integration with a BSP.

**driver method**

A driver method is a published entry point into a driver made available to an API in VxBus. Examples of methods include functionality such as connecting network interfaces to the MUX and discovery of interrupt routing. See also *method ID*.

**enumeration**

Enumeration 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 enumeration phase. For bus types such as VME, which do not have such functionality, tables that describe the devices that may be present on the system are maintained in the BSP. See also *bus discovery*.

**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*.

**mBlk**

Structure used to organize data buffers. See also *cluster*.

**method ID**

A *method ID* is the identification of a specific driver method that may be provided by a driver. This must be unique for each method (that is, specific functionality module) on the system. See also *driver method*.

**parameter**

Information about some aspect of device software configuration. For further discussion, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*. See also *resource*.

**parent**

The bus to which a device is attached, or the bus controller of that bus.

**probe**

See *enumeration* and *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 comparison. This routine is optional. If specified, it is normally safe and acceptable for the routine to simply indicate acceptance.

**processor Local bus (PLB)**

The bus connected directly to a processor. This term is used in a processor-agnostic way in this documentation.

**resource**

information about some aspect of device hardware configuration. For further discussion, see *VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals*. See also *parameter*.

**serial bitbang**

Serial bitbang describes a scenario where software writes the individual bits of a word out on a serial line, often with a corresponding clock, rather than writing the entire value into a register and allowing the underlying hardware to take care of the delivery of the word.

**service driver**

A device driver that provides a service to the operating system or to middleware, instead of a service for another device driver. Examples of service drivers include drivers for serial and network devices.

**stall**

A condition that occurs when a network interface device stops operating due to momentary lack of resources.

**upstream**

From the perspective of a device, upstream refers to a point closer to the CPU on the bus hierarchy. See also *parent*.

# Index

## Symbols


(*busCfgRead)( )
    overriding 9
(*busCfgWrite)( )
    overriding 9
(*dmaCancel)( ) 35
(*dmaPause)( ) 35
(*dmaRead)( ) 34
(*dmaReadAndWait)( ) 34
(*dmaStatus)( ) 36
(*dmaWrite)( ) 34
(*dmaWriteAndWait)( ) 35
(*timerAllocate)( ) 221
(*timerCountGet)( ) 223
(*timerCountGet64)( ) 227
(*timerDisable)( ) 224
(*timerEnable)( ) 224
(*timerEnable64)( ) 226
(*timerISRSet)( ) 225
(*timerRelease)( ) 222
(*timerRolloverGet)( ) 222
(*timerRolloverGet64)( ) 226
(*vxbDevControl)( )
    overriding 9
(*xf_dump)( ) 204, 207
(*xf_ioctl)( ) 204, 207, 209
(*xf_strategy)( ) 204, 207, 208
{busCtlrAccessOverride}( ) 8
{busCtlrBaseAddrCvt}( ) 11
{busCtlrCfgInfo}( ) 10
{busCtlrCfgRead}( ) 6
{busCtlrCfgWrite}( ) 7
{busDevShow}( ) 54
{cpmCommand}( ) 186
{driverControl}( ) 186, 246
{isrRerouteNotify}( ) 68
{m85xxLawBarAlloc}( ) 186
{miiBusRead}( ) 126
{miiLinkUpdate}( ) 92, 100, 127, 133
{miiMediaUpdate}( ) 90
{miiModeGet}( ) 100, 129
{miiModeSet}( ) 100, 128
{miiRead}( ) 90, 100, 127, 132, 133
{miiWrite}( ) 91, 127, 132, 133
{muxConnect}( ) 133
{muxDevConnect}( ) 88, 89, 101, 103
{nonVolGet}( ) 137
{nonVolSet}( ) 137
{sioChanConnect}( ) 191
{sioChanGet}( ) 191
{vxbDevRegMap}( ) 11
{vxbDmaResDedicatedGet}( ) 31
{vxbDmaResourceGet}( ) 30
{vxbDmaResourceRelease}( ) 31
{vxbDrvUnlink}( ) 90, 238
{vxbIntCtlrConnect}( ) 41
{vxbIntCtlrCpuReroute}( ) 44, 67
{vxbIntCtlrDisable}( ) 42

{vxbIntCtlrDisconnect}( ) 41
{vxbIntCtlrEnable}( ) 42
{vxbIntCtlrIntReroute}( ) 43
{vxbIntDynaVecConnect}( ) 43, 63
{vxbIntDynaVecProgram}( ) 17, 64
{vxbTimerFuncGet}( ) 215
{vxIpiControlGet}( ) 44, 69

# Numerics

00tffs.cdf 156

# A

access routine 249
accessing MII management registers 90
address translation
    DMA 98
    for USB drivers 238
    in bus controller drivers 11
advertise
    definition 249
advertising XBD methods 206
allocating
    device structures 78
    DMA channels 30
    system resources 186
    timers 215, 221
    tuples from the pool 98
AMD/Fujitsu flash devices
    CFI 145
    non-CFI 147
amdmtd.c 147
analog-to-digital converters 246
announcing
    a downstream bus 21
    devices to VxBus 22, 26, 78
ATA commands 210
ATA_RESOURCE 203
ATA_TYPE 203
ataResources 203
ataTypes 203
autoIntRouteSet 19
auxiliary clock 230

# B

BAM 143, 169
binding a network device to the stack 89
bio 204, 207, 208
bio.h 207
bio_done( ) 204, 208
block allocation map
    *see* BAM
boot block 145
bridgePostConfigFuncSet 19
bridgePreConfigFuncSet 19
BSP configuration
    bus controller drivers 17
    CPU routing table 49
    DMA drivers 32
    dynamic vector table 48
    dynamic vectors 63
    interrupt controller drivers 45
    interrupt input table 46
    interrupt priority 50
    MAC drivers 93
    multifunction drivers 77
    NVRAM drivers 138
    other class drivers 247
    PHY drivers 130
    resource drivers 187
    resources for a PCI bus 18
    serial drivers 193
    storage drivers 203
    timer drivers 218
    USB drivers 238
BSP resources
    ataResources 203
    ataTypes 203
    clkFreq 218
    clkRateMax 218
    clkRateMin 218
    cpuClkRate 218
    for PCI autoconfiguration 19
    inputTableSize 47

miiIfName 94
miiIfUnit 94
numSegments 138
phyAddr 93
priority 50
priorityTableSize 50
segments 138
buffer management 97, 104
BUFFER_WRITE_BROKEN 144
bus
definition 249
bus controller drivers 3
address translation 11
BSP configuration 17
communication 5
debugging 26
deferring driver initialization 27
driver methods 5
generating configuration transactions 10
header files 17
initialization 23
example 24
location in VxWorks 4
overriding service routines 8
overview 4
utility routines 20
bus controllers
connecting to devices 22
definition 249
bus discovery 249
bus match 250
bus type
definition 250
macros 22
registering with VxBus 25


## C


cacheSize 19
callbackInstall( ) 196
calling the socket registration routines 159
cancelling an operation on a DMA channel 35
cap_available 108
cap_enabled 108, 112
CDFs
00tffs.cdf 156
CFI 144
AMD/Fujitsu command set 144
AMD/Fujitsu flash devices 145
Intel/Sharp command set 144
CFI/SCS flash support 145
CFI_DEBUG 144
cfiamd.c 144
cfiscs.c 144, 145
changing the network link state 92
checksum offloading 107
CSUM flags 110
enabling
checksum for IPv6 TCP 110
checksum for IPv6 UDP 110
IP checksum 110
TCP checksum 110
UDP checksum 110
handling corrupt packets 113
interface capability flags 109
IPv4 receive 109
IPv4 transmit 109
IPv6 receive 109
IPv6 transmit 110
receive routine 112
send routine 114
child 250
classes
*see* driver classes
clkFreq 218
clkRateMax 218
clkRateMin 218
CLOCAL 195
cluster 104, 250
common flash interface
*see* CFI
communication, bus controller 5
components
adding MTD components 155
INCLUDE_GENERICPHY 126, 130
INCLUDE_GENERICTBIPHY 126, 130
INCLUDE_MII_BUS 123, 130
INCLUDE_NON_VOLATILE_RAM 138
INCLUDE_PARAM_SYS 130

config.h
MTD identification 157
configuring
BSP resources for PCI bus 18
interrupt controllers 52
connecting
an ISR to a timer 225
bus controllers to devices 22
ISRs 41
networking code 102
copying data to and from NVRAM 137
CPU routing table 49
cpuClkRate 218
CREAD 195
creating
a new bus 79
an MII bus instance 99
an XBD 205
buffer pools 98
crossbar routing table 50
CSIZE 195
CSUM flags 108, 114
CSUM_DATA_VALID 110, 113
CSUM_DELAY_DATA 111
CSUM_DELAY_DATA6 111
CSUM_DELAY_IP 111
csum_flags_rx 109
csum_flags_tx 109
CSUM_FRAGMENT 110
CSUM_IP 110
CSUM_IP_CHECKED 110, 112
CSUM_IP_FRAGS 110
CSUM_IP_HDRLEN 111
CSUM_IP_VALID 110, 112
CSUM_PSEUDO_HDR 111, 113
CSUM_RESULTS 111
CSUM_TCP 110
CSUM_TCP_SEG 110
CSUM_TCPv6 110
CSUM_TCPv6_SEG 110
CSUM_UDP 110
CSUM_UDPv6 110
CSUM_XPORT_CSUM_OFF 111
CSUM_XPORT_HDRLEN 111
custom drivers 246

# D

data structures
*see* structures
DEBUG_PRINT 145
debugging
bus controller drivers 26
DMA drivers 36
interrupt controller drivers 73
MAC drivers 118
multifunction drivers 82
NVRAM drivers 141
other class drivers 248
PHY drivers 133
resource drivers 188
serial drivers 199
storage drivers 211
timer drivers 232
USB drivers 242
deferring
driver registration for MAC drivers 118
ISRs 103
defining MTDs
as VxWorks components 155
in the socket driver file 156
deleting an MII bus 99
descriptor 250
devices
accessing MII management registers 90
AMD/Fujitsu flash, CFI 145
AMD/Fujitsu flash, non-CFI 147
CFI
creating an XBD 205
definition 250
enumeration 25
flash 135, 143
Intel 28F008 flash 146
Intel 28F016 flash 146
interactions on multifunction chips 79
interleaved registers 79
multiple devices on a single chip 75
NAND 147

network device initialization 101
PHY 99
device probing and discovery 124
registering with VxBus 25
shared resources 81
timers 214
devResourceGet( ) 139
digital-to-analog converters 246
direct memory access drivers
*see* DMA drivers
disabling
interrupt inputs 41
ISRs 42, 52
timer interrupt generation 224
discovering
devices 22, 26
PHY devices 124
disk-on-chip support 147
dispatching
interrupts 55
ISRs 59
DMA
engines 85, 106
managing 106
scatter-gather 105
support 98
DMA channel
cancelling an operation on 35
getting status of 36
pausing a 35
queueing a read operation 34
queueing a write operation 34
DMA drivers 29
BSP configuration 32
debugging 36
driver methods 30
header files 32
initialization 33
overview 30
structures and routines 33
utility routines 33
DMA_IDLE 36
DMA_NOT_USED 36
DMA_PAUSED 36
DMA_RUNNING 36
documentation
about 2
downstream 250
driver
definition 250
driver classes 1
bus controller 3
direct memory access 29
interrupt controller 38
multifunction 75
network 83
network interface (MAC) 86
NVRAM 136
other 245
PHY 123
resource 81, 185
serial 190
timer 214
USB 235
driver methods
{busCtlrAccessOverride}( ) 8
{busCtlrBaseAddrCvt}( ) 11
{busCtlrCfgInfo}( ) 10
{busCtlrCfgRead}( ) 6
{busCtlrCfgWrite}( ) 7
{busDevShow}( ) 54
{cpmCommand}( ) 186
{driverControl}( ) 186, 246
{isrRerouteNotify}( ) 68
{m85xxLawBarAlloc}( ) 186
{miiBusRead}( ) 126
{miiLinkUpdate}( ) 92, 100, 127, 133
{miiMediaUpdate}( ) 90
{miiModeGet}( ) 100, 129
{miiModeSet}( ) 100, 128
{miiRead}( ) 90, 100, 127, 132, 133
{miiWrite}( ) 91, 127, 132, 133
{muxConnect}( ) 133
{muxDevConnect}( ) 88, 89, 101, 103
{nonVolGet}( ) 137
{nonVolSet}( ) 137
{sioChanConnect}( ) 191
{sioChanGet}( ) 191
{vxbDevRegMap}( ) 11
{vxbDmaResDedicatedGet}( ) 31

{vxbDmaResourceGet}( ) 30
{vxbDmaResourceRelease}( ) 31
{vxbDrvUnlink}( ) 90, 238
{vxbIntCtlrConnect}( ) 41
{vxbIntCtlrCpuReroute}( ) 44, 67
{vxbIntCtlrDisable}( ) 42
{vxbIntCtlrDisconnect}( ) 41
{vxbIntCtlrEnable}( ) 42
{vxbIntCtlrIntReroute}( ) 43
{vxbIntDynaVecConnect}( ) 43, 63
{vxbIntDynaVecProgram}( ) 17, 64
{vxbTimerFuncGet}( ) 215
{vxIpiControlGet}( ) 44, 69
bus controller drivers 5
definition 251
DMA drivers 30
interrupt controller drivers 41
MAC drivers 88
multifunction drivers 76
multiprocessor systems 43
NVRAM drivers 137
other class drivers 246
PHY drivers 126
resource drivers 186
serial drivers 190
storage drivers 202
timer drivers 215
USB drivers 238
driverAccessFunc( ) 16
DRV_CTRL 25
dynamic vector assignment 43
dynamic vector table 48
dynamic vectors 40
configuring
in the BSP 63
using service driver routines 63
determining values for 65
interrupt assignment 17
programming 64

# E

EHCI 236
EIOCGIFCAP 108, 111
EIOCGIFMEDIA 100
EIOCSIFCAP 108, 111
enabling
interrupt inputs 42
IP checksum 110
ISRs 52
TCP checksum 110
timer interrupt generation 224
UDP checksum 110
encoding I/O operations 16
end.h 108
END_CAPABILITIES 108, 111, 112
END_ERR_LINKDOWN 92
END_ERR_LINKUP 92
END_MEDIALIST 101
END_OBJ 101
endian conversion for USB data transfers 240
endLib 98
endLib.c 97
endLoad( ) 101
endMedia.h 128, 129
endPoolCreate( ) 97
endPoolDestroy( ) 97, 98
endPoolJumboCreate( ) 97
endPoolTupleFree( ) 97, 98
endPoolTupleGet( ) 97, 98, 105
Enhanced Host Controller Interface
*see* EHCI
enhanced network drivers
*see* END drivers
enumeration 251
device 25, 26
erasableBlockSize 155
erase units 177
ERF 205
event reporting 208
event types 209
new device notification 207
registering 205
ERF_ASYNC_PROC 208
ERF_SYNC_PROC 208
erfEventRaise( ) 203, 207, 208
erfHandlerRegister( ) 203, 206
erfHandlerUnregister( ) 203, 206
Ethernet 85

event reporting framework
*see* ERF
extended block device
*see* XBD
EXTRA_INCLUDE 45

# F

fair received packet handling 116
fbbEnable 19
files
00tffs.cdf 156
amdmtd.c 147
cfiamd.c 144
cfiscs.c 144, 145
config.h 157
endLib.c 97
hwconf.c 45, 130
I28F008.c 146
i28f016.c 146
miiBus.c 99
miiBus.o 123
sysTffs.c 145, 146, 147, 155, 156, 157, 158
tffsConfig.c 149, 156
usbPciStub.c 239
vxbAuxClkLib.c 230
vxbIntCtlrLib.c 54
vxbIntelIchStorage.c 210
vxbPci.c 20
vxbSI31xxStorage.c 210
vxbSysClkLib.c 228
vxbTimestampLib.c 231
vxBus.c 78
finding interrupt inputs 53
flags, receive handler interlocking 115
flash
device layout 178
erase units 177
interleaved chips 177
flash file system support
*see* TrueFFS
flash translation layer
*see* FTL
FLASH_BASE_ADRS 161
FLASH_SIZE 161
flbase.h 151
flDelayMsec( ) 160
flDontNeedVpp( ) 153
FLFlash 148, 149, 155
flflash.h 149
flNeedVpp( ) 153
flSetWindowSize( ) 161
FLSocket 151, 159, 165
flSocketOf( ) 162
flsystem.h 154
flWriteProtected( ) 153
freeing a DMA channel 31
FTL 143, 166
overview 170
structures 171
terminology 166
function pointers
(*busCfgRead)( ) 9
(*busCfgWrite)( ) 9
(*dmaCancel)( ) 35
(*dmaPause)( ) 35
(*dmaRead)( ) 34
(*dmaReadAndWait)( ) 34
(*dmaStatus)( ) 36
(*dmaWrite)( ) 34
(*dmaWriteAndWait)( ) 35
(*timerAllocate)( ) 221
(*timerCountGet)( ) 223
(*timerCountGet64)( ) 227
(*timerDisable)( ) 224
(*timerEnable)( ) 224
(*timerEnable64)( ) 226
(*timerISRSet)( ) 225
(*timerRelease)( ) 222
(*timerRolloverGet)( ) 222
(*timerRolloverGet64)( ) 226
(*vxbDevControl)( ) 9
(*xf_dump)( ) 204, 207
(*xf_ioctl)( ) 204, 207, 209
(*xf_strategy)( ) 204, 207, 208
vxbIntDynaCtlrInputInit( ) 56
vxbIntDynaVecProgram( ) 56

# G

generating
  bus controller configuration transactions 10
  dynamic vectors 55
genericPhy 123, 124, 125, 129
genericTbiPhy 126
genericTbiPhy.h 126
getting
  an ISR function pointer 53
  arguments for an ISR 53
  flags for an interrupt input 53
  status of a DMA channel 36

# H

handling
  interrupts 103
  network interrupts 88
HCDs 236, 237
HCF_RES_ADDR 18, 47
HCF_RES_INT 18, 19, 47
header files
  bio.h 207
  bus controller drivers 17
  DMA drivers 32
  end.h 108
  endMedia.h 128, 129
  flbase.h 151
  flflash.h 149
  flsystem.h 154
  genericTbiPhy.h 126
  interrupt controller drivers 45
  MAC drivers 92
  multifunction drivers 77
  NVRAM drivers 137
  other class drivers 247
  PHY drivers 130
  resource drivers 187
  serial drivers 192
  sioLib.h 190, 194
  sioLibCommon.h 195
  storage drivers 202, 218
  USB drivers 238
  usbHst.h 238
  usbOsal.h 238
  vxbAccess.h 8, 14
  vxbDmaDriverLib.h 31
  vxbDmaLib.h 33
  vxbIntCtlrLib.h 45
  vxbIntrCtlr.h 45
  vxbNonVol.h 137
  vxbTimerLib.h 215
  vxBus.h 22, 77
  xbd.h 207
heartbeat interrupt 214
hEND drivers
  *see* hierarchical END drivers
HEND_RX_QUEUE_PARAM 97
hierarchical END drivers 134
host controller drivers
  *see* HCDs
HUPCL 195
hwconf.c 52, 130
  interrupt controller resources 45
  NVRAM 138
  USB drivers 239
hwMemAlloc( ) 25, 220

# I

I28F008.c 146
i28f016.c 146
identifying interrupts 39
IFCAP_CAP0 110
IFCAP_CAP1 110
IFCAP_CAP2 110
IFCAP_IPCOMP 110
IFCAP_IPSEC 109
IFCAP_JUMBO_MTU 109
IFCAP_NETCONS 109
IFCAP_RXCSUM 109, 112
IFCAP_RXCSUMv6 109
IFCAP_TCPSEG 109
IFCAP_TCPSEGv6 110
IFCAP_TXCSUM 109
IFCAP_TXCSUMv6 110
IFCAP_VLAN_HWTAGGING 109

IFCAP_VLAN_MTU 109
ifconfig( ) 119, 121
IFM_ACTIVE 129
IFM_AUTO 101, 132
IFM_AVALID 129
implementing
  timer driver service routines 221
  VxWorks auxiliary clock 230
  VxWorks system clock 228
  VxWorks timestamp drivers 231
INCLUDE_EHCI 242
INCLUDE_GENERICPHY 126, 130
INCLUDE_GENERICTBIPHY 126, 130
INCLUDE_MII_BUS 123, 130
INCLUDE_MTD_AMD 147
INCLUDE_MTD_CFISCS 145
INCLUDE_MTD_I28F008 147
INCLUDE_MTD_I28F016 146
INCLUDE_NON_VOLATILE_RAM 138
INCLUDE_OHCI 242
INCLUDE_PARAM_SYS 130
INCLUDE_UHCI 242
INCLUDE_USB_INIT 241
includeFuncSet 19
initializing
  a network 119
  bus controller drivers 23
  DMA drivers 33
  FLFlash structure members 149
  interrupt controller drivers 56
  MAC drivers 101
  multifunction drivers 79
  network devices 101
  NVRAM drivers 139
  other class drivers 248
  PHY drivers 132
  resource drivers 187
  serial drivers 193, 199
  storage drivers 204
  timer drivers 219
  USB drivers 241
inputTableSize 47
instance
  definition 251
intAssignFuncSet 19
intCpuUnlock( ) 60
intCtlrChainISR( ) 53, 55
intCtlrCpu 49
intCtlrHwConfGet( ) 51, 52
intCtlrHwConfShow( ) 51, 53
intCtlrISRAdd( ) 51, 52
intCtlrISRDisable( ) 51, 52
intCtlrISREnable( ) 51, 52
intCtlrISRRemove( ) 51, 53
intCtlrPinFind( ) 51, 53, 63
intCtlrStrayISR( ) 53, 55
intCtlrTableArgGet( ) 51, 53
intCtlrTableCreate( ) 51, 54
intCtlrTableFlagsGet( ) 51, 53
intCtlrTableFlagsSet( ) 51, 54
intCtlrTableIsrGet( ) 51, 53, 55
intCtlrTableUserSet( ) 51, 54
integrating a timer driver with VxWorks 228
Intel 28F008 flash devices 146
Intel 28F016 flash devices 146
Intel ICH storage driver 205
interaction
  PHY and MII bus 99
  serial ports and WDB connection 198
interleaved registers 79
INTERLEAVED_MODE_REQUIRES_32BIT_WRITES 144
interprocessor interrupts
  *see* IPIs
interrupt controller drivers 38
  BSP configuration 45
  CPU routing table 49
  crossbar routing table 50
  debugging 73
  dispatch routines 55
  driver methods 41
  dynamic vector assignment 40, 43
  dynamic vector table 48
  header files 45
  initialization 56
  interrupt input table 46
  interrupt priority 50, 58
  managing dynamic vectors 62
  multiprocessing 40, 43, 66
  OpenPIC 38

overview 38
programming dynamic vectors 64
releasing third-party drivers 45
responsibilities 39
typologies 57
utility routines 51
vxbEpicIntCtlr.c 38
vxbPpcIntCtlr.c 38
interrupt controllers
configuration 39, 52
layers 59
interrupt inputs 39
representing internally 65
interrupt-driven mode
serial drivers 194
interrupts
connecting ISRs 52
disabling
interrupt inputs 41
ISRs 52
dispatching 55
dynamic vector assignment 17, 40, 43
dynamic vector management 55
dynamic vector table 48
enabling 55
interrupt inputs 42
ISRs 52
finding inputs 53
getting
an ISR function pointer for 53
flags 53
handlers 103
identifying 39
input table 46
interrupt controller drivers 38
managing dynamic vectors 62
network 88
PHY 123
priority 50, 58
programming dynamic vectors 64
removing an ISR 53
rerouting 43, 44
retrieving ISR arguments 53
routing in an SMP system 67
serial drivers 199
setting flags in isrHandle 54
transmit-packet-complete 87
validating 120
intrCtlrInputs 46
intrCtlrPriority 50
io16Addr 18, 19
io16Size 19
io32Addr 18, 19
io32Size 18, 19
ioctl( ) 195
iodesc 16
ipcom_drv_eth_init( ) 119
IPIs 68
managing 44
IPsec 109
isrDeferIsrReroute( ) 68
isrDeferLib 68
isrHandle 52, 53, 65
printing contents of 53
setting flags in 54
isrRerouteNotify( ) 68
ISRs 103
calling 54
connecting 41
to an interrupt 52
to timer hardware 225
deferring 103
disabling 42, 52
dispatching 59
enabling 52
function pointers 53
removing 53
retrieving arguments to 53

## J

jobQueueCreate( ) 97
jobQueueInit( ) 97
jobQueueLib 94, 103
jobQueuePost( ) 96, 103, 115, 117
jobQueueStdPost( ) 97
jumbo frames, support for 97

# L

libraries
endLib 98
isrDeferLib 68
jobQueueLib 94, 103
muxLib 94, 95
netBufLib 94, 97, 104
utility library for PCI configuration 18
vxbAuxClkLib 230
vxbDmaBufLib 94, 98, 106
vxbDmaLib 30, 33
vxbIntCtlrLib 39, 51, 55, 65
vxbSysClkLib 228
vxbTimestampLib 231
vxIpiLib 69
logMsg( ) 120
lower edge methods 127
lower edge utility routines 131

# M

MAC drivers 83
attaching
to the IPv4 stack 119
to the MUX 119
binding a device to the stack 89
BSP configuration 93
command and control module 87
connecting networking code 102
debugging 118
with show routines 118
deferring driver registration 118
driver methods 88
fair received packet handling 116
functional modules 86
handling checksum offload 107
header files 92
initialization 101
interrupt handlers 103
loading and unloading 121
lower edge methods 127
lower edge utility routines 131
multicast filter test 122
overview 86
pairing with a PHY instance 119
PHY and MII bus interactions 99
ping-of-death test 120
polled mode testing 122
protocol impact on 107
receive error path testing 122
receive handler interlocking flag 115
receive stall handling 117
reception module 87
relationship to MII bus 125
setting up a memory pool 104
starting and stopping 121
stress testing 120
support for scatter-gather 105
terminating an instance 90
testing with Netperf 120
transmission module 87
upper edge methods 127
upper edge utility routines 130
utility routines 94
validating interrupts 120
WTX test 122
macros
BUFFER_WRITE_BROKEN 144
CFI_DEBUG 144
DEBUG_PRINT 145
INCLUDE_EHCI 242
INCLUDE_MTD_AMD 147
INCLUDE_MTD_CFISCS 145
INCLUDE_MTD_I28F008 147
INCLUDE_MTD_I28F016 146
INCLUDE_OHCI 242
INCLUDE_UHCI 242
INTERLEAVED_MODE_REQUIRES_32BIT_WRITES 144
SAVE_NVRAM_REGION 144
VXB_BUSID_MII 22
VXB_BUSID_PCI 22
VXB_BUSID_PLB 22
VXB_BUSID_RAPIDIO 22
VXB_BUSID_VIRTUAL 22
VXB_HANDLE( ) 16
VXB_HANDLE_WIDTH( ) 16
VXB_INTCTLR_ISR_CALL( ) 51, 54, 55, 60

VXB_INTCTLR_PINENTRY
_ALLOCATED( ) 51, 55, 67
VXB_INTCTLR_PINENTRY
_ENABLED( ) 51, 55
VXB_INTCTLRLIB_LOWLVL_SIZE 65
VXB_INTCTLRLIB_TOPLVL_SIZE 65
managing
dynamic vectors 48, 62
system resources 186
mapping
device registers 11
maxBusSet 19
maxLatAllSet 19
maxLatencyArgSet 19
maxLatencyFuncSet 19
mBlk 251
mBlkPktHdr 108
MDIO 91
media access controller
*see* MAC drivers
media independent interface
*see* MII
mem32Addr 18, 19
mem32Size 18, 19
memIo32Addr 18, 19
memIo32Size 18, 19
memory technology driver
*see* MTDs
message signalled interrupt
*see* MSI
method ID 251
MII 88
MII bus 123
creating 99
deleting 99
interactions with PHY devices 99
lower edge methods 127
lower edge utility routines 131
management 99
relationship to MAC 125
upper edge methods 127
upper edge utility routines 130
miiBus.c 99
miiBus.o 123
miiBusCreate( ) 91, 99, 125, 133
miiBusDelete( ) 99
miiBusDevMatch( ) 124
miiBusMediaAdd( ) 131
miiBusMediaDefaultSet( ) 131
miiBusMediaDel( ) 131
miiBusMediaListGet( ) 101, 131
miiBusModeGet( ) 92, 100, 127
miiBusModeSet( ) 100, 127
miiBusMonitor 92, 99, 123, 127, 133
miiBusRead( ) 132
miiBusWrite( ) 132
miiIfName 94
miiIfUnit 94
miiMonitor 90
msgLogSet 19
MSI 48, 62
MSI-X 62
MTDs 142
customizing 143
defining
as components 155
in the socket driver file 156
erase routine 155
helper routines 153
initializing the FLFlash
structure members 149
non-CFI 146
read routine 153
registering an identification routine 156
supported flash devices 143
write routine 154
writing 148
a map routine 152
read, write, and erase routines 153
the identification routine 148
mtdTable[ ] 149, 156
MTU
jumbo 109
VLAN-compatible 109
multifunction drivers 75
BSP configuration 77
debugging 82
device interconnections 79
driver methods 76
header files 77

initialization 79
interleaved registers 79
location of subordinate devices 81
overview 76
reducing footprint 77
shared resources 81
utility routines 78
multiplexor
*see* MUX
multiprocessor systems
CPU routing table 49
interrupt controller drivers 40
limitations 72
routing interrupts in 67
MUX 85, 95, 102, 107
attaching to 119
muxDevLoad( ) 95, 101, 121
muxDevStart( ) 96
muxDevStop( ) 96, 121
muxDevUnload( ) 96, 121
muxError( ) 90, 92
muxIoctl( ) 96
muxLib 94, 95
muxSend( ) 87
muxTxRestart( ) 96


## N


NAND devices 147
necEndLoad( ) 89
netBufLib 94, 97, 104
netJobQueueId 96
Netperf test suite 120
netPoolCreate( ) 98
netPoolRelease( ) 98
netTupleFree( ) 98
netTupleGet( ) 98
network console 109
network drivers 83
*see also* MAC drivers, PHY drivers
hierarchical END drivers (hEND) 134
interrupts 88
protocols 85
terminology 84
wireless Ethernet drivers 133
network interface drivers
*see* MAC drivers
network processing tasks 96
networking 97
deferring ISRs 103
fair received packet handling 116
handling checksum offload 107
interrupt handlers 103
link state changes 92
overview 84
ping-of-death 120
protocol impact on drivers 107
receive handler interlocking flag 115
receive stall handling 117
setting up a memory pool 104
supporting scatter-gather 105
transmission media 85
NIC_DRV_CTRL 89
non-volatile RAM drivers
*see* NVRAM drivers
notifying the ERF of a new device 207
numSegments 138
NVRAM
block sizes 139
copying data to 137
stacking instances 140
NVRAM drivers 135
BSP configuration 138
debugging 141
driver methods 137
header files 137
initialization 139
overview 136
utility routines 139
nvRamSegment 138


## O


OHCI 236
Open Host Controller Interface
*see* OHCI
Open Systems Interconnection (OSI) 84

operating system abstraction layer
*see* OSAL
operating voltage 163
OS_MALLOC( ) 238
OSAL 238
other class drivers 245
BSP configuration 247
debugging 248
driver methods 246
header files 247
initialization 248
overview 246
utility routines 248
overriding
(*busCfgRead)( ) 9
(*busCfgWrite)( ) 9
(*vxbDevControl)( ) 9
bus controller service routines 8
register access routines 12

# P

pairing a MAC driver with a PHY device 119
parallel drivers 202
*see also* storage drivers
parameter
definition 251
PARENB 195
parent 251
PARODD 195
pausing a DMA channel 35
PCI bus
autoconfiguration 19
BSP resources 18
configuration 18
utility routines for PCI autoconfiguration 21
utility routines for PCI configuration 20
pciConfigMechanism 19
pciIo16Size 18
PCMCIA socket drivers 158
pDrvCtrl 89, 116
periodic interrupt timer drivers 214
*see also* timer drivers
required service routines 221
PHY drivers 83, 99, 123
BSP configuration 130
debugging 133
device probing and discovery 124
driver methods 126
generic PHY driver support 125
generic TBI driver support 126
genericPhy 123, 124, 125, 129
genericTbiPhy 126
header files 130
initialization 132
lower edge methods 127
lower edge utility routines 131
MAC and MII bus relationship 125
overview 123
pairing with a MAC driver 119
upper edge methods 127
upper edge utility routines 130
utility routines 130
phyAddr 93
PLB 4
definition 252
polled mode
serial drivers 194
pollInput( ) 197
pollOutput( ) 197
powerOnCallback 151
printing isrHandle contents 53
priority 50
priorityTableSize 50
probe 251
probe routine 252
processor local bus
*see* PLB
programming dynamic vectors 64
programming voltage 163
protocols
impact on drivers 107

# Q

queueing
DMA read operations 34
DMA write operations 34

# R


reading and writing device registers 11
reading bus configuration space 6
receive handler interlocking flag 115
receive stall handling 117
reducing footprint for multifunction drivers 77
register access 11
  creating a handle for transaction types 13
  handle values for access types 14
  PHY 123
  predefined transaction types 13
  providing a new transaction type 15
registering
  an MTD identification routine 156
  devices and bus types with VxBus 25
  with the ERF 205
registers
  interleaved 79
releasing
  a buffer pool 98
  a DMA channel 31
  a timer 222
removing
  a device from the system 78
  an ISR from isrHandle 53
rerouting interrupts to a specified CPU 43, 44
resident flash array
  *see* RFA
resource
  definition 252
resource drivers 81, 185
  BSP configuration 187
  debugging 188
  driver methods 186
  header files 187
  initialization 187
  overview 186
  utility routines 187
resources
  *see* BSP resources
returning
  a device structure to the pool 26, 78
  a tuple to the pool 98
RFA 158, 159
rfaCardDetected( ) 160, 163
rfaGetAndClearChangeIndicator( ) 162, 165
rfaRegister( ) 159, 162
rfaSetMappingContext( ) 161, 164
rfaSetWindow( ) 159, 161, 164
rfaSocketInit( ) 159, 161, 164
rfaVccOff( ) 160, 163
rfaVccOn( ) 160, 163
rfaVppOff( ) 160, 164
rfaVppOn( ) 160, 163
rfaWriteProtected( ) 162, 165
ring buffer
  overflowing 104
rollcallFuncSet 19
root hub class drivers 237
routines
  (*dmaCancel)( ) 35
  (*dmaPause)( ) 35
  (*dmaRead)( ) 34
  (*dmaReadAndWait)( ) 34
  (*dmaStatus)( ) 36
  (*dmaWrite)( ) 34
  (*dmaWriteAndWait)( ) 35
  (*timerAllocate)( ) 221
  (*timerCountGet)( ) 223
  (*timerCountGet64)( ) 227
  (*timerDisable)( ) 224
  (*timerEnable)( ) 224
  (*timerEnable64)( ) 226
  (*timerISRSet)( ) 225
  (*timerRelease)( ) 222
  (*timerRolloverGet)( ) 222
  (*timerRolloverGet64)( ) 226
  (*xf_dump)( ) 204, 207
  (*xf_ioctl)( ) 204, 207, 209
  (*xf_strategy)( ) 204, 207, 208
  bio_done( ) 204, 208
  callbackInstall( ) 196
  devResourceGet( ) 139
  DMA 33
  driverAccessFunc( ) 16
  endLoad( ) 101
  endPoolCreate( ) 97
  endPoolDestroy( ) 97, 98
  endPoolJumboCreate( ) 97

endPoolTupleFree( ) 97, 98
endPoolTupleGet( ) 97, 98, 105
erfEventRaise( ) 203, 207, 208
erfHandlerRegister( ) 203, 206
erfHandlerUnregister( ) 203, 206
flDelayMsec( ) 160
flDontNeedVpp( ) 153
flNeedVpp( ) 153
flSetWindowSize( ) 161
flSocketOf( ) 162
flWriteProtected( ) 153
for configuring dynamic vectors 63
hwMemAlloc( ) 25, 220
ifconfig( ) 119, 121
intCpuUnlock( ) 60
intCtlrChainISR( ) 53, 55
intCtlrHwConfGet( ) 51, 52
intCtlrHwConfShow( ) 51, 53
intCtlrISRAdd( ) 51, 52
intCtlrISRDisable( ) 51, 52
intCtlrISREnable( ) 51, 52
intCtlrISRRemove( ) 51, 53
intCtlrPinFind( ) 51, 53, 63
intCtlrStrayISR( ) 53, 55
intCtlrTableArgGet( ) 51, 53
intCtlrTableCreate( ) 51, 54
intCtlrTableFlagsGet( ) 51, 53
intCtlrTableFlagsSet( ) 51, 54
intCtlrTableIsrGet( ) 51, 53, 55
intCtlrTableUserSet( ) 51, 54
ioctl( ) 195
ipcom_drv_eth_init( ) 119
isrDeferIsrReroute( ) 68
isrRerouteNotify( ) 68
jobQueueCreate( ) 97
jobQueueInit( ) 97
jobQueuePost( ) 96, 103, 115, 117
jobQueueStdPost( ) 97
logMsg( ) 120
miiBusCreate( ) 91, 99, 125, 133
miiBusDelete( ) 99
miiBusDevMatch( ) 124
miiBusMediaAdd( ) 131
miiBusMediaDefaultSet( ) 131
miiBusMediaDel( ) 131
miiBusMediaListGet( ) 101, 131
miiBusModeGet( ) 92, 100, 127
miiBusModeSet( ) 100, 127
miiBusRead( ) 132
miiBusWrite( ) 132
MTD helper routines 153
muxDevLoad( ) 95, 101, 121
muxDevStart( ) 96
muxDevStop( ) 96, 121
muxDevUnload( ) 96, 121
muxError( ) 90, 92
muxIoctl( ) 96
muxSend( ) 87
muxTxRestart( ) 96
necEndLoad( ) 89
netPoolCreate( ) 98
netPoolRelease( ) 98
netTupleFree( ) 98
netTupleGet( ) 98
OS_MALLOC( ) 238
pollInput( ) 197
pollOutput( ) 197
rfaCardDetected( ) 160, 163
rfaGetAndClearChangeIndicator( ) 162, 165
rfaRegister( ) 159, 162
rfaSetMappingContext( ) 161, 164
rfaSetWindow( ) 159, 161, 164
rfaSocketInit( ) 159, 161, 164
rfaVccOff( ) 160, 163
rfaVccOn( ) 160, 163
rfaVppOff( ) 160, 164
rfaVppOn( ) 160
rfaWriteProtected( ) 162, 165
setWindow( ) 161
stackTxRestartRtn( ) 96
sysAuxClkConnect( ) 230
sysAuxClkDisable( ) 230
sysAuxClkEnable( ) 230
sysAuxClkRateGet( ) 230
sysAuxClkRateSet( ) 230
sysClkConnect( ) 228
sysClkDisable( ) 228
sysClkEnable( ) 228
sysClkRateGet( ) 228
sysClkRateSet( ) 228

sysHwInit( ) 229, 231, 232
sysTffsInit( ) 157, 159
sysTimestamp( ) 231
sysTimestampConnect( ) 231
sysTimestampDisable( ) 231
sysTimestampEnable( ) 231
sysTimestampFreq( ) 231
sysTimestampLock( ) 231
sysTimestampPeriod( ) 231
taskDelay( ) 117
txStartup( ) 196
usbBusToMem( ) 239
usbInit( ) 241
usbMemToBus( ) 239
usbMemToPci( ) 239
usbPciToMem( ) 239
usrEnableCpu( ) 67
usrNetInit( ) 119
vxbBusAnnounce( ) 21, 25, 79
vxbDeviceAnnounce( ) 26, 78
vxbDevIterate( ) 247
vxbDevRemovalAnnounce( ) 78
vxbDevStructAlloc( ) 26, 78
vxbDevStructFree( ) 26, 78
vxbDmaChanAlloc( ) 31, 33
vxbDynaIntConnect( ) 52
vxbIntConnect( ) 39, 52, 54, 57, 63
vxbIntCtlrPinEntryGet( ) 67
vxbIntDynaConnect( ) 63
vxbIntDynaCtlrInputInit( ) 56
vxbIntDynaVecProgram( ) 56
vxbIntToCpuRoute( ) 67
vxbMsiConnect( ) 63
vxbNonVolGet( ) 137
vxbNonVolSet( ) 137, 140
vxbPciAutoConfig( ) 19, 21
vxbPciBusTypeInit( ) 22
vxbPciConfigLibInit( ) 20
vxbPciDeviceAnnounce( ) 22, 25
vxbPciMSIProgram( ) 64
vxbRead*( ) 11
vxbRead16( ) 11
vxbRead32( ) 11
vxbRead64( ) 11
vxbRead8( ) 11
vxbRegMap( ) 12
vxbWrite*( ) 12
vxbWrite16( ) 12
vxbWrite32( ) 12
vxbWrite64( ) 12
vxbWrite8( ) 12
xbdAttach( ) 204, 206, 208
rxQueue00 96

# S

SATA drivers 202
*see also* storage drivers
SAVE_NVRAM_REGION 144, 145
scatter-gather 105
segmentation
IPv4/TCP 109
IPv6/TCP 110
segments 138
serial ATA drivers
*see* SATA drivers
serial bitbang 252
serial drivers 190
BSP configuration 193
debugging 199
driver methods 190
header files 192
initialization 193, 198, 199
interaction between serial ports and WDB connection 198
interrupts 198, 199
overview 190
polled versus interrupt-driven mode 194
utility routines 193
WDB 197, 198
service driver 252
setPowerOnCallback 151
setting up a memory pool 104
setWindow( ) 161
seven layer OSI model 84
shared resources on multifunction chips 81

show routines
debugging MAC drivers 118
intCtlrHwConfShow( ) 51
printing isrHandle contents 53
vxBusShow( ) 82
Silicon Image storage driver 205
SIO_AVAIL_MODES_GET 196
SIO_BAUD_GET 195
SIO_BAUD_SET 195
SIO_CALLBACK_GET_TX_CHAR 196
SIO_CALLBACK_PUT_RCV_CHAR 196
SIO_CHAN 190, 194
SIO_CHANNEL_INFO 191
SIO_DRV_FUNCS 194
callbackInstall( ) 196
ioctl( ) 195
pollInput( ) 197
pollOutput( ) 197
txStartup( ) 196
SIO_HUP 196
SIO_HW_OPTS_GET 195
SIO_HW_OPTS_SET 195
SIO_MODE_GET 196
SIO_MODE_SET 196, 197
SIO_OPEN 196
sioLib.h 190, 194
sioLibCommon.h 195
SMP 40
SMP considerations
for interrupt controller drivers 66
timer drivers 233
socket driver file
*see* sysTffs.c
socket drivers 157
calling the socket registration routines 159
implementing the member function structure 159
multiple drivers 161
PCMCIA 158, 162
required functionality 162
RFA 158
stub file 158
SOCKET_12_VOLTS 163
sockets
address mapping 165
member functions 163
registration 162
windowing 165
spinlocks
using with driverAccessFunc( ) 16
stacking NVRAM instances 140
stackTxRestartRtn( ) 96
stall 252
storage drivers 201
ATA commands 210
BSP configuration 203
debugging 211
driver methods 202
event reporting 208
header files 202
initialization 204
Intel ICH 205
interface with VxWorks file systems 205
overview 202
processing queued work 208
Silicon Image 205
utility routines 203
writing new drivers 210
structures
ATA_RESOURCE 203
ATA_TYPE 203
bio 204, 207, 208
DRV_CTRL 25
END_CAPABILITIES 108, 111, 112
END_MEDIALIST 101
END_OBJ 101
FLFlash 148, 149, 155
FLSocket 151, 159, 165
for DMA drivers 33
FTL 171
HEND_RX_QUEUE_PARAM 97
intCtlrCpu 49
intrCtlrInputs 46
isrHandle 65
mBlkPktHdr 108
nvRamSegment 138
pDrvCtrl 89, 116
SIO_CHAN 190, 194

SIO_CHANNEL_INFO 191
SIO_DRV_FUNCS 194
timer drivers 219
vol 159
VXB_ACCESS_LIST 8
VXB_DMA_REQUEST 31
vxbDevRegInfo 23
vxbDmaFuncs 33
vxbDmaResource 33
vxbPciConfig 10
vxbTimerFunctionality 221, 226, 230, 232
VXIPI_CTRL_INIT 44, 69
xbd_funcs 204, 207
supporting scatter-gather 105
symmetric multiprocessing
*see* SMP
sysAuxClkConnect( ) 230
sysAuxClkDisable( ) 230
sysAuxClkEnable( ) 230
sysAuxClkRateGet( ) 230
sysAuxClkRateSet( ) 230
sysClkConnect( ) 228
sysClkDisable( ) 228
sysClkEnable( ) 228
sysClkRateGet( ) 228
sysClkRateSet( ) 228
sysHwInit( ) 229, 231, 232
system clock 228
sysTffs.c 145, 146, 147, 155, 156, 157, 158
sysTffsInit( ) 157, 159
sysTimestamp( ) 231
sysTimestampConnect( ) 231
sysTimestampDisable( ) 231
sysTimestampEnable( ) 231
sysTimestampFreq( ) 231
sysTimestampLock( ) 231
sysTimestampPeriod( ) 231


## T


taskDelay( ) 117
tasks 92
miiBusMonitor 99, 123, 127, 133
miiMonitor 90
tNet0 96, 127
TBI 126
TBI_ID1 126
TBI_ID2 126
TCP/IP, checksum offloading 107
terminating an instance 90
terminology
for network drivers 84
terms
access routine 249
advertise 249
bus 249
bus controller 249
bus discovery 249
bus match 250
bus type 250
child 250
cluster 250
descriptor 250
device 250
downstream 250
driver 250
driver method 251
enumeration 251
instance 251
mBlk 251
method ID 251
parameter 251
parent 251
PLB 252
probe 251
probe routine 252
resource 252
serial bitbang 252
service driver 252
stall 252
upstream 252
testing
MAC drivers
loading and unloading 121
multicast filter test 122
ping-of-death 120

polled mode 122
receive error path 122
starting and stopping 121
stress testing 120
with Netperf 120
WTX test 122
tffsConfig.c 149, 156
timer drivers 214
allocating a timer 221
BSP configuration 218
connecting an ISR to a timer 225
data structure layout 219
debugging 232
disabling timer interrupt generation 224
driver methods 215
enabling timer interrupt generation 224
getting
the current value of a timer 223
the maximum return value for a timer 222
header files 218
implementing service routines 221
initialization 219
integrating with VxWorks 228
overview 214
releasing a timer 222
SMP considerations 233
sysHwInit( ) 229, 231, 232
utility routines 219
VxWorks auxiliary clock 230
VxWorks system clock 228
timers
64-bit 216, 226, 227
connecting an ISR to 225
disabling interrupt generation on 224
enabling interrupt generation on 224
timestamp drivers 214
*see also* timer drivers
implementing 231
required service routines 221
tNet0 96, 127
tNetTask
*see* tNet0
transmit-packet-complete interrupt 87
True Flash File System
*see* TrueFFS
TrueFFS 135, 141
driver development 143
erase units 177
layers
core layer 142
flash translation layer (FTL) 143, 166
MTD layer 142
socket layer 142
overview 141
socket drivers 157
tuple 97
txQueue00 96
txStartup( ) 196

# U

UHCI 236
Universal Host Controller Interface
*see* UHCI
universal serial bus
*see* USB
upper edge methods 127
upper edge utility routines 130
upstream 252
USB 236
On-The-Go 236
USB drivers 235
address conversion routines 240
address translation 238
BSP configuration 238
debugging 242
driver methods 238
endian conversion for data transfers 240
header files 238
host controller drivers (HCDs) 237
hwconf.c 239
INIT macros 242
initialization 241
example 242
initializing
class drivers 241
USB host controller devices 241
non-VxBus 236
OSAL 238

overview 236
peripheral stack drivers 237
registering with VxBus 241
root hub class 237
USB host stack drivers 236
utility routines 241
VxBus model 236
usbBusToMem( ) 239
USBD 236
interface 238
usbHst.h 238
usbInit( ) 241
usbMemToBus( ) 239
usbMemToPci( ) 239
usbOsal.h 238
usbPciStub.c 239
usbPciToMem( ) 239
usrEnableCpu( ) 67
usrNetInit( ) 119
utility routines
bus controller drivers 20
DMA drivers 33
interrupt controller drivers 51
MAC drivers 94
multifunction drivers 78
NVRAM drivers 139
other class drivers 248
PCI autoconfiguration 21
PCI configuration 20
PHY drivers 130
resource drivers 187
serial drivers 193
storage drivers 203
timer drivers 219
USB drivers 241


# V


validating interrupts 120
VBM
*see* virtual block map
Vcc 163
vector 62
assigned 62
dynamic 62
virtual block map 168
VLAN
hardware tags 109
vol 159
Vpp 163
VXB_ACCESS_LIST 8
VXB_BUSID_MII 22, 124
VXB_BUSID_PCI 22
VXB_BUSID_PLB 22
VXB_BUSID_RAPIDIO 22
VXB_BUSID_VIRTUAL 22
VXB_DEVID_BUSCTRL 23
VXB_DEVID_DEVICE 23
VXB_DMA_REQUEST 31
VXB_DMA_RESOURCE_ID 33
VXB_HANDLE( ) 16
VXB_HANDLE_IO 14
VXB_HANDLE_MEM 14
VXB_HANDLE_ORDERED 14
VXB_HANDLE_SWAP 14
VXB_HANDLE_WIDTH( ) 16
VXB_INTCTLR_ISR_CALL( ) 51, 54, 55, 60
VXB_INTCTLR_PINENTRY
_ALLOCATED( ) 51, 55, 67
VXB_INTCTLR_PINENTRY_ENABLED( ) 51, 55
VXB_INTCTLR_SPECIFIC_1 54
VXB_INTCTLR_SPECIFIC_2 54
VXB_INTCTLRLIB_LOWLVL_SIZE 65
VXB_INTCTLRLIB_TOPLVL_SIZE 65
VXB_INTR_DYNAMIC 48, 62, 63
VXB_TIMER_AUTO_RELOAD 216
VXB_TIMER_CAN_INTERRUPT 216
VXB_TIMER_CANNOT_DISABLE 216
VXB_TIMER_CANNOT_MODIFY
_ROLLOVER 216
VXB_TIMER_CANNOT_SUPPORT_ALL
_FREQS 216
VXB_TIMER_INTERMEDIATE_COUNT 216, 233
VXB_TIMER_SIZE_16 216
VXB_TIMER_SIZE_23 216
VXB_TIMER_SIZE_32 216
VXB_TIMER_SIZE_64 216, 226

VXB_TIMER_STOP_WHILE_READ 216
vxbAccess.h 8, 14
vxbAuxClkLib 230
vxbAuxClkLib.c 230
vxbBusAnnounce( ) 21, 25, 79
vxbDeviceAnnounce( ) 26, 78
vxbDevIterate( ) 247
vxbDevRegInfo 23
vxbDevRemovalAnnounce( ) 78
vxbDevStructAlloc( ) 26, 78
vxbDevStructFree( ) 26, 78
vxbDmaBufLib 94, 98, 106
vxbDmaChanAlloc( ) 31, 33
vxbDmaDriverLib.h 31
vxbDmaFuncs 33
vxbDmaLib 30, 33
vxbDmaLib.h 33
vxbDmaResource 33
vxbDynaIntConnect( ) 52
VxbEnd drivers
    *see* network drivers
vxbEpicIntCtlr.c 38
vxbIntConnect( ) 39, 52, 54, 57, 63
vxbIntCtlrLib 39, 51, 55, 65
vxbIntCtlrLib.c 54
vxbIntCtlrLib.h 45
vxbIntCtlrPinEntryGet( ) 67
vxbIntDynaConnect( ) 63
vxbIntDynaCtlrInputInit( ) 56
vxbIntDynaVecProgram( ) 56
vxbIntelIchStorage.c 210
vxbIntrCtlr.h 45
vxbIntToCpuRoute( ) 67
vxbMsiConnect( ) 63
vxbNonVol.h 137
vxbNonVolGet( ) 137
vxbNonVolSet( ) 137, 140
vxbPci.c 20
vxbPciAutoConfig( ) 19, 21
vxbPciBusTypeInit( ) 22
vxbPciConfig 10
vxbPciConfigLibInit( ) 20
vxbPciDeviceAnnounce( ) 22, 25
vxbPciMSIProgram( ) 64
vxbPpcIntCtlr.c 38
vxbRead*( ) 11
vxbRead16( ) 11
vxbRead32( ) 11
vxbRead64( ) 11
vxbRead8( ) 11
vxbRegMap( ) 12
vxbSI31xxStorage.c 210
vxbSysClkLib 228
vxbSysClkLib.c 228
vxbTimerFunctionality 221, 226, 230, 232
vxbTimerLib.h 215
vxbTimestampLib 231
vxbTimestampLib.c 231
vxBus.c 78
vxBus.h 22, 77
vxBusShow( ) 82
vxbWrite*( ) 12
vxbWrite16( ) 12
vxbWrite32( ) 12
vxbWrite64( ) 12
vxbWrite8( ) 12
VXIPI_CTRL_INIT 44, 69
vxIpiLib 69
vxPci.c 18
VxWorks SMP
    *see* SMP

# W

WDB 198
    kernel initialization 197
    serial drivers 197
WDB_COMM_SERIAL 197
WDB_COMM_TYPE 197
Wind River USB
    *see* USB
wireless Ethernet drivers 133
writing
    drivers for multifunction devices 75
    MTD components 148
    MTD identification routine 148
    MTD map routine 152
    MTD read, write, and erase routines 153
    new storage drivers 210

socket drivers for TrueFFS 157
to bus configuration space 7

# X

XBD 205
advertising methods 206
creating 205
event types 209
xbd.h 207
xbd_funcs 204, 207
XBD_HARD_EJECT 209
XBD_SOFT_EJECT 209
XBD_STACK_COMPLETE 209
xbdAttach( ) 204, 206, 208
xbdEventInstantiated 209
xbdEventMediaChanged 209
xbdEventPrimaryInsert 209
xbdEventRemove 209