1 Class-Specific Driver Development .......................................................... 1
   1.1 About VxBus Driver Classes ................................................................. 1
   1.2 Before You Begin ................................................................................... 1
   1.3 About this Document ............................................................................ 2
       Navigating this Documentation Set ....................................................... 2

2 Bus Controller Drivers ............................................................................ 3
   2.1 Introduction ............................................................................................ 3
   2.2 Overview .................................................................................................. 3
   2.3 VxBus Driver Methods ............................................................................ 5
       2.3.1 {busCtrlDevCfgRead}() ..................................................................... 5
       2.3.2 {busCtrlCfgRead}() .......................................................................... 6
       2.3.3 {busCtrlDevCfgWrite}() .................................................................. 7
       2.3.4 {busCtrlCfgWrite}() ........................................................................ 7
       2.3.5 {busCtrlDevCtrl}() .......................................................................... 8
       2.3.6 {busCtrlAccessOverride}() .............................................................. 9
         Override for (*busCfgRead)() .............................................................. 10
         Override for (*busCfgWrite)() .............................................................. 10
         Override for (*vxbDevControl)() ......................................................... 10
       2.3.7 {busCtrlCfgInfo}() ......................................................................... 11
       2.3.8 {busCtrlBaseAddrCvt}() .................................................................. 11
       2.3.9 {vxbDevRegMap}() .......................................................................... 12
         Specifying a Predefined Transaction Type ......................................... 13
         Providing a New Transaction Type ..................................................... 15
       2.3.10 {vxbIntDynaVecProgram}() ............................................................ 16
   2.4 Header Files ........................................................................................... 16
2.5 BSP Configuration ........................................................................................................... 17
  2.5.1 PCI Configuration ....................................................................................................... 17
  2.5.2 PCI Autoconfiguration ............................................................................................... 18

2.6 Available Utility Routines ............................................................................................... 19
  2.6.1 PCI Configuration ....................................................................................................... 19
  2.6.2 PCI Autoconfiguration ............................................................................................... 19
  2.6.3 vxbBusAnnounce( ) ................................................................................................. 20
  2.6.4 vxBpciBusTypeInit( ) .............................................................................................. 20

2.7 Initialization .................................................................................................................... 21
  2.7.1 Initialization Example ............................................................................................... 22
    vxbBusAnnounce( ) ........................................................................................................ 23
    vxbDeviceAnnounce( ) .................................................................................................. 23
    vxbDevStructAlloc( ) ................................................................................................. 23
    vxbDevStructFree( ) ................................................................................................. 24

2.8 Debugging ....................................................................................................................... 24

3 Direct Memory Access Drivers ......................................................................................... 25
  3.1 Introduction ................................................................................................................... 25
  3.2 Overview ....................................................................................................................... 25
  3.3 VxBus Driver Methods .................................................................................................... 26
    3.3.1 {vxbDmaResourceGet}( ) ..................................................................................... 26
    3.3.2 {vxbDmaResourceRelease}( ) ............................................................................. 27
    3.3.3 {vxbDmaResDedicatedGet}( ) .............................................................................. 27

3.4 Header Files ................................................................................................................... 27

3.5 BSP Configuration .......................................................................................................... 28

3.6 Available Utility Routines .............................................................................................. 28

3.7 Initialization .................................................................................................................... 28

3.8 DMA System Structures and Routines .......................................................................... 28
  3.8.1 (*dmaRead)( ) .................................................................................................... 29
  3.8.2 (*dmaReadAndWait)( ) .................................................................................... 29
  3.8.3 (*dmaWrite)( ) .................................................................................................. 29
  3.8.4 (*dmaWriteAndWait)( ) .................................................................................. 29
  3.8.5 (*dmaCancel)( ) ............................................................................................. 30
  3.8.6 (*dmaPause)( ) ............................................................................................... 30
  3.8.7 (*dmaResume)( ) ............................................................................................ 30
3.8.8 (*dmaStatus)( ) ................................................................................................... 30

3.9 Debugging ................................................................................................................. 31

4 Interrupt Controller Drivers ........................................................................ 33

4.1 Introduction ............................................................................................................... 33

4.2 Overview .................................................................................................................. 34

    Interrupt Identification .................................................................................... 34
    Interrupt Controller Driver Responsibilities ................................................. 34
    Interrupt Controller Configurations ............................................................... 35
    Dynamic Vectors ................................................................................................ 35
    Interrupt Controller Drivers and Multiprocessing ....................................... 36

4.3 VxBus Driver Methods .......................................................................................... 36

    4.3.1 Basic Methods ........................................................................................... 36

    |vxbIntCtlrConnect() ...................................................................................... 36
    |vxbIntCtlrDisconnect() ................................................................................. 36
    |vxbIntCtlrEnable() ........................................................................................ 37
    |vxbIntCtlrDisable() ........................................................................................ 37

    4.3.2 Dynamic Vector Methods ........................................................................ 37

    |vxbIntDynaVecConnect() ............................................................................. 37

    4.3.3 Multiprocessor Methods .......................................................................... 38

    |vxbIntCtlrIntReroute() .................................................................................. 38
    |vxbIntCtlrCpuReroute() ............................................................................... 38
    |vxIpiControlGet() .......................................................................................... 39

4.4 Header Files ........................................................................................................... 39

    vxbIntrCtlr.h ....................................................................................................... 39
    vxbIntCtlrLib.h .................................................................................................. 39

4.5 BSP Configuration ................................................................................................. 40

    4.5.1 Interrupt Input Table ............................................................................... 40
    4.5.2 Dynamic Vector Table ............................................................................. 42
    4.5.3 CPU Routing Table ................................................................................... 42
    4.5.4 Interrupt Priority ....................................................................................... 43
    4.5.5 Crossbar Routing Table .......................................................................... 43

4.6 Available Utility Routines .................................................................................... 44

    4.6.1 intCtlrHwConfGet( ) ................................................................................ 45
    4.6.2 intCtlrISRAdd( ) ....................................................................................... 45
    4.6.3 intCtlrISRDisable( ) ............................................................................... 45
    4.6.4 intCtlrISREnable( ) .................................................................................. 45
    4.6.5 intCtlrISRRemove( ) ............................................................................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.6</td>
<td>intCtlrPinFind()</td>
</tr>
<tr>
<td>4.6.7</td>
<td>intCtlrTableArgGet()</td>
</tr>
<tr>
<td>4.6.8</td>
<td>intCtlrTableFlagsGet()</td>
</tr>
<tr>
<td>4.6.9</td>
<td>intCtlrTableIsrGet()</td>
</tr>
<tr>
<td>4.6.10</td>
<td>intCtlrHwConfShow()</td>
</tr>
<tr>
<td>4.6.11</td>
<td>intCtlrTableCreate()</td>
</tr>
<tr>
<td>4.6.12</td>
<td>intCtlrTableFlagsSet()</td>
</tr>
<tr>
<td>4.6.13</td>
<td>intCtlrTableUserSet()</td>
</tr>
<tr>
<td>4.6.14</td>
<td>VXB_INTCTLR_ISR_CALL()</td>
</tr>
<tr>
<td>4.6.15</td>
<td>VXB_INTCTLR_PINENTRY_ENABLED()</td>
</tr>
<tr>
<td>4.6.16</td>
<td>VXB_INTCTLR_PINENTRY_ALLOCATED()</td>
</tr>
<tr>
<td>4.6.17</td>
<td>Dispatch Routines</td>
</tr>
<tr>
<td></td>
<td>vxbIntDynaCtlrInputInit()</td>
</tr>
<tr>
<td></td>
<td>vxbIntDynaVecProgram()</td>
</tr>
<tr>
<td>4.7</td>
<td>Initialization</td>
</tr>
<tr>
<td>4.8</td>
<td>Interrupt Controller Typologies and Hierarchies</td>
</tr>
<tr>
<td>4.9</td>
<td>Interrupt Priority</td>
</tr>
<tr>
<td>4.10</td>
<td>ISR Dispatch</td>
</tr>
<tr>
<td>4.11</td>
<td>Managing Dynamic Interrupt Vectors</td>
</tr>
<tr>
<td></td>
<td>Configuring Dynamic Vectors Using the Service Driver Routines</td>
</tr>
<tr>
<td></td>
<td>Configuring Dynamic Vectors in the BSP</td>
</tr>
<tr>
<td></td>
<td>Programming Dynamic Vectors</td>
</tr>
<tr>
<td></td>
<td>Determining Dynamic Vector Values</td>
</tr>
<tr>
<td>4.12</td>
<td>Internal Representation of Interrupt Inputs</td>
</tr>
<tr>
<td>4.13</td>
<td>Multiprocessor Issues with VxWorks SMP</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Routing Interrupt Inputs to Individual CPUs</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Interprocessor Interrupts</td>
</tr>
<tr>
<td>4.13.3</td>
<td>Limitations in Multiprocessor Systems</td>
</tr>
<tr>
<td>4.14</td>
<td>Debugging</td>
</tr>
<tr>
<td>5</td>
<td>Multifunction Drivers</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview</td>
</tr>
<tr>
<td>5.3</td>
<td>VxBus Driver Methods</td>
</tr>
<tr>
<td>5.4</td>
<td>Header Files</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Initialization for Network Interface Drivers</td>
</tr>
<tr>
<td>6.2.7</td>
<td>MUX: Connecting to Networking Code</td>
</tr>
<tr>
<td>6.2.8</td>
<td>jobQueueLib: Deferring ISRs</td>
</tr>
<tr>
<td>6.2.9</td>
<td>Working with Ipcom_pkt Packets</td>
</tr>
<tr>
<td>6.2.10</td>
<td>Supporting Scatter-Gather with IPNET-Native Drivers</td>
</tr>
<tr>
<td>6.2.11</td>
<td>Protocol Impact on Drivers</td>
</tr>
<tr>
<td>6.2.12</td>
<td>Other Network Interface Driver Issues</td>
</tr>
<tr>
<td></td>
<td>Receive Handling Method</td>
</tr>
<tr>
<td></td>
<td>Receive Stall Handling</td>
</tr>
<tr>
<td>6.2.13</td>
<td>Debugging Network Interface Drivers</td>
</tr>
<tr>
<td></td>
<td>Using VxBus Show Routines</td>
</tr>
<tr>
<td></td>
<td>Deferring Driver Registration</td>
</tr>
<tr>
<td></td>
<td>Pairing with a PHY instance</td>
</tr>
<tr>
<td></td>
<td>Stress Testing</td>
</tr>
<tr>
<td></td>
<td>Netperf Test Suite</td>
</tr>
<tr>
<td></td>
<td>Interrupt Validation</td>
</tr>
<tr>
<td></td>
<td>Additional Tests</td>
</tr>
<tr>
<td>6.3</td>
<td>PHY Drivers</td>
</tr>
<tr>
<td>6.3.1</td>
<td>PHY Driver Overview</td>
</tr>
<tr>
<td></td>
<td>PHY Device Probing and Discovery</td>
</tr>
<tr>
<td></td>
<td>MAC and MII Bus Relationship</td>
</tr>
<tr>
<td></td>
<td>Generic PHY Driver Support</td>
</tr>
<tr>
<td></td>
<td>Generic TBI Driver Support</td>
</tr>
<tr>
<td>6.3.2</td>
<td>VxBus Driver Methods for PHY Drivers</td>
</tr>
<tr>
<td></td>
<td>Upper Edge Methods</td>
</tr>
<tr>
<td></td>
<td>Lower Edge Methods</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Header Files for PHY Drivers</td>
</tr>
<tr>
<td>6.3.4</td>
<td>BSP Configuration for PHY Drivers</td>
</tr>
<tr>
<td>6.3.5</td>
<td>Available Utility Routines for PHY Drivers</td>
</tr>
<tr>
<td></td>
<td>Upper Edge Utility Routines</td>
</tr>
<tr>
<td></td>
<td>Lower Edge Utility Routines</td>
</tr>
<tr>
<td>6.3.6</td>
<td>Initialization for PHY Drivers</td>
</tr>
<tr>
<td>6.3.7</td>
<td>Debugging PHY Drivers</td>
</tr>
<tr>
<td>6.4</td>
<td>Wireless Ethernet Drivers</td>
</tr>
<tr>
<td>6.5</td>
<td>Hierarchical END Drivers</td>
</tr>
</tbody>
</table>
### 7 Non-Volatile RAM Drivers

#### 7.1 Introduction
NVRAM Drivers and TrueFFS

#### 7.2 Non-Volatile RAM Drivers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVRAM Driver Overview</td>
<td>138</td>
</tr>
<tr>
<td>VxBus Driver Methods for NVRAM Drivers</td>
<td>138</td>
</tr>
<tr>
<td>{nonVolGet}()</td>
<td>138</td>
</tr>
<tr>
<td>{nonVolSet}()</td>
<td>138</td>
</tr>
<tr>
<td>Header Files</td>
<td>139</td>
</tr>
<tr>
<td>BSP Configuration for NVRAM Drivers</td>
<td>139</td>
</tr>
<tr>
<td>Utility Routines for NVRAM Drivers</td>
<td>140</td>
</tr>
<tr>
<td>Initialization for NVRAM Drivers</td>
<td>140</td>
</tr>
<tr>
<td>NVRAM Block Sizes</td>
<td>140</td>
</tr>
<tr>
<td>Stacking NVRAM Instances</td>
<td>141</td>
</tr>
<tr>
<td>Debugging NVRAM Drivers</td>
<td>141</td>
</tr>
</tbody>
</table>

#### 7.3 Flash File System Support with TrueFFS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrueFFS Overview</td>
<td>142</td>
</tr>
<tr>
<td>Core Layer</td>
<td>142</td>
</tr>
<tr>
<td>MTD Layer</td>
<td>142</td>
</tr>
<tr>
<td>Socket Layer</td>
<td>142</td>
</tr>
<tr>
<td>Flash Translation Layer</td>
<td>142</td>
</tr>
<tr>
<td>TrueFFS Driver Development Process</td>
<td>143</td>
</tr>
<tr>
<td>Using MTD-Supported Flash Devices</td>
<td>143</td>
</tr>
<tr>
<td>Writing MTD Components</td>
<td>146</td>
</tr>
<tr>
<td>Socket Drivers</td>
<td>154</td>
</tr>
<tr>
<td>Flash Translation Layer</td>
<td>160</td>
</tr>
</tbody>
</table>

### 8 Resource Drivers

#### 8.1 Introduction

#### 8.2 Overview

#### 8.3 VxBus Driver Methods

#### 8.4 Header Files

#### 8.5 BSP Configuration

#### 8.6 Available Utility Routines

#### 8.7 Initialization

#### 8.8 Debugging
10.8 Interface with VxWorks File Systems ................................................................. 192
  10.8.1 Device Creation ............................................................................................. 192
  ERF Registration ................................................................................................. 192
  Advertisement of XBD Methods ...................................................................... 193
  ERF New Device Notification ......................................................................... 194
  10.8.2 Processing .................................................................................................... 194
  10.8.3 Event Reporting .......................................................................................... 195
10.9 Writing New Storage Drivers ............................................................................. 196
10.10 Debugging ........................................................................................................ 197
11 Timer Drivers ..................................................................................................... 199
  11.1 Introduction ..................................................................................................... 199
  11.2 Overview ......................................................................................................... 199
  11.3 VxBus Driver Methods ..................................................................................... 200
  11.4 Header Files .................................................................................................... 203
  11.5 BSP Configuration .......................................................................................... 203
  11.6 Available Utility Routines .............................................................................. 203
  11.7 Initialization ..................................................................................................... 203
  11.8 Data Structure Layout .................................................................................... 204
  11.9 Implementing Driver Service Routines .......................................................... 205
    11.9.1 (*timerAllocate)() ..................................................................................... 205
    11.9.2 (*timerRelease)() ..................................................................................... 205
    11.9.3 (*timerRolloverGet)() ............................................................................. 206
    11.9.4 (*timerCountGet)() ................................................................................. 206
    11.9.5 (*timerDisable)() ..................................................................................... 207
    11.9.6 (*timerEnable)() ........................................................................................ 208
    11.9.7 (*timerISRSet)() ..................................................................................... 208
    11.9.8 (*timerEnable64)() .................................................................................. 209
    11.9.9 (*timerRolloverGet64)() .......................................................................... 209
    11.9.10 (*timerCountGet64)() ........................................................................... 210
  11.10 Integrating a Timer Driver ............................................................................ 211
    11.10.1 VxWorks System Clock .......................................................................... 211
    11.10.2 VxWorks Auxiliary Clock ....................................................................... 213
    11.10.3 VxWorks Timestamp Driver ................................................................... 214
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.
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.

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 to see actual implementations and to understand how bus controller drivers interact with VxWorks. These drivers are located in:

```
installDir/vxworks-6.x/target/src/hwif/busCtlr
```

### 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 `{busCtlrDevCfgRead}()`

The `{busCtlrDevCfgRead}()` 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 `{busCtlrDevCfgRead}()` method is implemented using a driver-provided routine with the following prototype:

```c
LOCAL STATUS func{busCtlrDevCfgRead}( 
    VXB_DEVICE_ID pInst, /* device info */
    PCI_HARDWARE * pPciDev, /* PCI device info */
    UINT32 byteOffset, /* offset into config space */
    UINT32 transactionSize, /* transaction size, in bytes */
    void * pDstBuf /* buffer to write to */
)
```

The parameters to `func{busCtlrDevCfgRead}()` are:

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

- **pPciDev**
  The PCI device info for 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.

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

**NOTE:** As of VxWorks 6.7 (VxBus version 4.0.0), the \{busCtlrCfgRead\}( ) is no longer supported for bus controller drivers and is replaced by the \{busCtlrDevCfgRead\}( ) method (see 2.3.1 \{busCtlrDevCfgRead\}( ), p.5). If you have an existing driver that makes use of this method, you must define the VXB\_LEGACY\_ACCESS macro and recompile the source in:

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

For more information, see the VxBus version considerations section of the VxWorks Device Driver Developer’s Guide (Vol. 1): Device Driver Fundamentals.

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.
2.3.3 \{busCtlrDevCfgWrite\}()

The \{busCtlrDevCfgWrite\}() 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 \{busCtlrDevCfgWrite\}() method is implemented using a driver-provided routine with the following prototype:

```c
LOCAL STATUS func{busCtlrDevCfgWrite}(
   VXB_DEVICE_ID pInst, /* device info */
   PCI_HARDWARE * pPciDev, /* PCI device info */
   UINT32 byteOffset, /* offset into config space */
   UINT32 transactionSize, /* transaction size, in bytes */
   UINT32 data /* data write to the offset */
)
```

The parameters to `func{busCtlrDevCfgWrite}()` are:

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

- **pPciDev**
  The PCI device info for 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.

- **data**
  The data that will be written to configuration space.

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

2.3.4 \{busCtlrCfgWrite\}()

**NOTE:** As of VxWorks 6.7 (VxBus version 4.0.0), the \{busCtlrCfgWrite\}() is no longer supported for bus controller drivers and is replaced by the \{busCtlrDevCfgWrite\}() method (see 2.3.3 \{busCtlrDevCfgWrite\}(), p.7). If you have an existing driver that makes use of this method, you must define the VXB_LEGACY_ACCESS macro and recompile the source in:

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

For more information, see the VxBus version considerations section of the VxWorks Device Driver Developer's Guide (Vol. 1): Device Driver Fundamentals.

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 \texttt{busCtlrCfgWrite}() method is implemented using a driver-provided routine with the following prototype:

\begin{verbatim}
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 */
)
\end{verbatim}

The parameters to \texttt{func{busCtlrCfgWrite}()} are:

\texttt{pInst}

The VXB\_DEVICE\_ID for the bus controller instance.

\texttt{bus}

The PCI bus number of the target hardware.

\texttt{dev}

The PCI device number of the target hardware.

\texttt{func}

The PCI function number of the target hardware.

\texttt{byteOffset}

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

\texttt{transactionSize}

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

\texttt{pSrcBuf}

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

\texttt{pFlags}

Reserved.

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

\subsection{2.3.5 \texttt{busCtlrDevCtlr}()}

The \texttt{busCtlrDevCtlr}() method is used to handle manipulation of downstream devices, such as interrupt management. Currently, this method is used exclusively on the PCI bus by the PCI bus support code.

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

\begin{verbatim}
LOCAL STATUS func{busCtlrDevCtlr}(
    VXB_DEVICE_ID pInst, /* device info */
    pVXB\_DEVCTL\_HDR pBusDevControl /* access identifier */
)
\end{verbatim}
The parameters to func{busCtlrDevCfgWrite}() are:

pInst
The VXB_DEVICE_ID for the bus controller instance.

pBusDevControl
A pointer to the structure that holds the access identifier and forms the first member of the structure.

2.3.6 {busCtlrAccessOverride}()

NOTE: As of VxWorks 6.7 (VxBus 4.0.0), the {busCtlrAccessOverride}() is no longer supported for bus controller drivers. The functionality provided by this method is not required in VxBus 4.0.0. If you have an existing driver that makes use of this method, you must define the VXB_LEGACY_ACCESS macro and recompile the source in:

installDir/vxworks-6.x/target/src

For more information, see the VxBus version considerations section of the VxWorks Device Driver Developer’s Guide (Vol. I): Device Driver Fundamentals.

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. The VXB_ACCESS_LIST structure is declared in the following file:

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:

```c
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:

```c
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:

```c
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 2.3.9 {vxbDevRegMap}( ), p.12.
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).

2.3.7 {busCtrlCfgInfo}()

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 {busCtrlCfgInfo}() 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 {busCtrlCfgInfo}() method is implemented using a driver-provided routine with the following prototype:

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

The implementation of {busCtrlCfgInfo}() 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.19.)

2.3.8 {busCtrlBaseAddrCvt}()

The {busCtrlBaseAddrCvt}() 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 {busCtrlBaseAddrCvt}() method is implemented using a driver-provided routine with the following prototype:

```
LOCAL STATUS func{busCtrlBaseAddrCvt}
{
    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 {busCtrlBaseAddrCvt}() 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.

### 2.3.9 `{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()`  

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:

```c
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)
2 Bus Controller Drivers
2.3 VxBus Driver Methods

- 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.
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>
<thead>
<tr>
<th>Type of Access</th>
<th>Value for Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory access</td>
<td>VXB_HANDLE_MEM</td>
</tr>
<tr>
<td>Ordered memory access</td>
<td>VXB_HANDLE_ORDERED</td>
</tr>
<tr>
<td>I/O access</td>
<td>VXB_HANDLE_IO</td>
</tr>
<tr>
<td>Byte-swapped memory access</td>
<td>VXB_HANDLE_SWAP(VXB HANDLE_MEM)</td>
</tr>
<tr>
<td>Byte-swapped order memory access</td>
<td>VXB_HANDLE_SWAP(VXB HANDLE ORDERED)</td>
</tr>
<tr>
<td>Byte-swapped I/O access</td>
<td>VXB_HANDLE_SWAP(VXB HANDLE_IO)</td>
</tr>
</tbody>
</table>

The preprocessor macros that are used to create the handle values are found in the following file:

```
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:

```c
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:

```c
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:

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

The bus controller access routine has the following prototype:

```c
LOCAL VOID driverAccessFunc
    (int iodesc, /* a descriptor for the requested I/O 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:

- **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.

These macros are found in:

`installDir/vxworks-6.x/target/src/hwif/h/vxbus/vxbAccess.h`
VxWorks
Device Driver Developer's Guide, 6.8

VXB_HANDLE_OP() can be used as follows:

```c
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.

### 2.3.10 `vxbIntDynaVecProgram`()

Provides support for dynamic interrupt vector assignment.

```c
STATUS func(vxbIntDynaVecProgram)
    VXBDEVICE_ID instID,
    struct vxbIntDynaVecInfo * dynaVec
```

For more information on dynamic vectors, see *Programming Dynamic Vectors*, p.54.

### 2.4 Header Files

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

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

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

```c
#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 resource entries become evident. At present, the majority of configuration resource entries 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCF_RES_ADDR</td>
<td>mem32Addr</td>
<td>Specifies the 32-bit pre-fetchable memory pool base address.</td>
</tr>
<tr>
<td>HCF_RES_INT</td>
<td>mem32Size</td>
<td>Specifies the 32-bit pre-fetchable memory pool size.</td>
</tr>
<tr>
<td>HCF_RES_ADDR</td>
<td>memIo32Addr</td>
<td>Specifies the 32-bit non-prefetchable memory pool base address.</td>
</tr>
<tr>
<td>HCF_RES_INT</td>
<td>memIo32Size</td>
<td>Specifies the 32-bit non-prefetchable memory pool size.</td>
</tr>
<tr>
<td>HCF_RES_ADDR</td>
<td>io32Addr</td>
<td>Specifies the 32-bit I/O pool base address.</td>
</tr>
<tr>
<td>HCF_RES_INT</td>
<td>io32Size</td>
<td>Specifies the 32-bit I/O pool size.</td>
</tr>
<tr>
<td>HCF_RES_ADDR</td>
<td>io16Addr</td>
<td>Specifies the 16-bit I/O pool base address.</td>
</tr>
<tr>
<td>HCF_RES_INT</td>
<td>io16Size</td>
<td>Specifies the 16-bit I/O pool size.</td>
</tr>
</tbody>
</table>

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 the following file:

`installDir/vxworks-6.x/target/src/hwif/vxbus/vxbPci.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.

### 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()`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HCF_RES_INT</td>
<td>pciConfigMechanism</td>
<td>A value between 0 and 2.</td>
</tr>
<tr>
<td></td>
<td>io16Addr</td>
<td>maxLatencyFuncSet</td>
</tr>
<tr>
<td></td>
<td>io16Size</td>
<td>mem32Addr</td>
</tr>
<tr>
<td></td>
<td>io32Addr</td>
<td>mem32Size</td>
</tr>
<tr>
<td></td>
<td>io32Size</td>
<td>memIo32Addr</td>
</tr>
<tr>
<td></td>
<td>maxBusSet</td>
<td>memIo32Size</td>
</tr>
<tr>
<td></td>
<td>maxLatAllSet</td>
<td>msgLogSet</td>
</tr>
<tr>
<td></td>
<td>maxLatencyArgSet</td>
<td>rolcallFuncSet</td>
</tr>
</tbody>
</table>

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 the following file:

\texttt{installDir/vxworks-6.x/target/src/hwif/vxbus/vxbPci.c}

To initialize the library, the bus controller driver must call \texttt{vxbPciConfigLibInit( )}. The prototype for \texttt{vxbPciConfigLibInit( )} is as follows:

\begin{verbatim}
STATUS vxbPciConfigLibInit
{
    struct vxbPciConfig *pPciConfig,
    int     pciMaxBus /* Max number of sub-busses */
}
\end{verbatim}

\textbf{NOTE:} The arguments \texttt{pciConfAddr0}, \texttt{pciConfAddr1}, \texttt{pciConfAddr2}, and \texttt{pciConfigMech} have been removed in this release. The \texttt{vxbPci} library only supports configuration method 0 (see 2.5.1 PCI Configuration, p.17).

\texttt{vxbPciConfigLibInit( )} initializes the memory pointed to by \texttt{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 \texttt{(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 \texttt{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:

```c
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.18, 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:

```c
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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXB_BUSID_PLB</td>
<td>Processor Local Bus</td>
</tr>
<tr>
<td>VXB_BUSID_PCI</td>
<td>PCI</td>
</tr>
<tr>
<td>VXB_BUSID_RAPIDIO</td>
<td>RapidIO</td>
</tr>
<tr>
<td>VXB_BUSID_MII</td>
<td>Media Independent Interface (MII)</td>
</tr>
<tr>
<td>VXB_BUSID_VIRTUAL</td>
<td>virtual bus</td>
</tr>
</tbody>
</table>

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 and to make the discovered devices on the bus visible to VxBus. The prototype for `vxbPciBusTypeInit()` is:

```c
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()`.

**NOTE:** The `vxbPciDeviceAnnounce()` routine available in VxWorks 6.6 and earlier releases is integrated into `vxbPciBusTypeInit()` in this release and is no longer available.

## 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.24.

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 Section 2.6 Available Utility Routines, p.19.

### 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.
   ```c
   pDrvCtrl = hwMemAlloc (sizeof(*pDrvCtrl));
   ```

2. Query required resources from the BSP, and store them locally.
   ```c
   devResourceGet(pHcf, "maxBusSet", HCF_RES_INT, (void *)&pDrvCtrl->pciMaxBus);
   /* etc. */
   ```

3. Initialize the PCI interrupt handling information.
   ```c
   vxbPciIntLibInit (pDrvCtrl->pIntInfo);
   ```

4. Initialize the support library used for PCI configuration handling.
   ```c
   vxbPciConfigLibInit (pDrvCtrl->pPciConfig, /*...*/);
   ```

5. Initialize the bus controller hardware itself.
   ```c
   g64120aPciBridgeInit (pInst);
   ```

6. Inform VxBus about the availability of the new bus.
   ```c
   vxbBusAnnounce (pInst, VXB_BUSID_PCI);
   ```

7. If the BSP has requested PCI autoconfiguration, perform the autoconfigure now.
   ```c
   if (pDrvCtrl->autoConfig)
     vxbPciAutoConfig (pInst);
   ```

8. Complete VxBus initialization.
   ```c
   vxbPciBusTypeInit (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. 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.

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

`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:

```c
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()`.

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

---

1. PCI Bus controller drivers call `vxbPciBusTypeInit()`, 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.
**vxbDevStructFree()**

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

```c
define 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*. 
## 3.1 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:

```c
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`

The structure is defined as follows:

```c
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:

```c
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 \texttt{pChan}. \texttt{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 \texttt{pReqDev}, or by checking information passed in using the \texttt{pDedicatedChanInfo} member of \texttt{pReq}. The prototype is as follows:

```c
STATUS \{vxbDmaResDedicatedGet\}
    (VXB_DEVICE_ID pReqDev,
     VXB_DMA REQUEST * pReq)
```

### 3.4 Header Files

DMA drivers must include the following header files:

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

Other drivers that wish to use \texttt{vxbDmaLib} may need to include the following:

```c
#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:

```c
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.

```c
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.

```c
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.

```c
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.
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.

```c
STATUS (*dmaCancel)(
    VXB_DMARESOURCE_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)().

```c
STATUS (*dmaPause)(
    VXB_DMARESOURCE_ID dmaChan
);
```

3.8.7 (*dmaResume)()

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

```c
STATUS (*dmaResume)(
    VXB_DMARESOURCE_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.

```c
int (*dmaStatus)(
    VXB_DMARESOURCE_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*. 
4.1 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:** When intended for use with the optional VxWorks SMP product, Wind River strongly recommends that the interrupt controller code 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, `vxbPpcIntCrtl.r` 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.55.

### 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 `vxbIntCtrlLib` 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, `vxbIntCtrlLib` 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.36.

**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.49.

**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.52.
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.56.

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.

```c
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.
4.3 VxBus Driver Methods

4.3.1 Static Vector Methods

The `vxbIntCtlrDisconnect` function disconnects an interrupt source from an interrupt controller.

```c
LOCAL STATUS 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 */
}
```

The `vxbIntCtlrEnable` function enables the interrupt input and marks the specified ISR as enabled.

```c
LOCAL STATUS vxbIntCtlrEnable
{
    VXBDEVICE_ID pIntCtlr, /* interrupt controller VxBus device ptr */
    VXBDEVICE_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 */
}
```

The `vxbIntCtlrDisable` function marks the specified ISR as disabled. If there are no other enabled ISRs chained to the same interrupt input, the routine disables the input.

```c
LOCAL STATUS 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.

The `vxbIntDynaVecConnect` function 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.

```c
LOCAL STATUS vxbIntDynaVecConnect
{
    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 */
}
```
When called, the \texttt{vxbIntDynaVecConnect()} routine causes interrupt vectors to be assigned to the requested device and connects the specified ISRs and arguments to the interrupts.

\begin{verbatim}
LOCAL STATUS func(vxbIntDynaVecConnect)
{
    VXB_DEVICE_ID pIntCtlr,
    VXB_DEVICE_ID pInst,
    int vecCount,
    struct vxbIntDynaVecInfo* dynaVec
}
\end{verbatim}

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 \textit{4.5 BSP Configuration}, p.40.

### 4.3.3 Multiprocessor Methods

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

\{vxbIntCtlrIntReroute\}()

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

\begin{verbatim}
LOCAL STATUS func(vxbIntCtlrIntReroute)
{
    VXB_DEVICE_ID pInst,
    int index,
    cpuset_t destCpu
}
\end{verbatim}

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

\begin{verbatim}
LOCAL STATUS func(vxbIntCtlrCpuReroute)
{
    VXB_DEVICE_ID pInst,
    void * destCpu
}
\end{verbatim}

While \texttt{vxbIntCtlrIntReroute()} is specific to a single interrupt input, \texttt{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 \texttt{hwconf.c}, then a single call to \texttt{vxbIntCtlrCpuReroute()} must reroute all four of those interrupts that are routed to CPU 1.
Interprocessor interrupts (IPIs) are used for various purposes in multiprocessor systems. The \texttt{func(vxIpiControlGet)}() routine returns a pointer to a structure, \texttt{VXIPI_CTRL_INIT}, containing information to manage IPIs.

\begin{verbatim}
LOCAL VXIPI_CTRL_INIT * func(vxIpiControlGet)
{
    VXB_DEVICE_ID pInst,
    void * pArg
}
\end{verbatim}

For more information on IPIs, see \textit{4.13.2 Interprocessor Interrupts}, p.57.

\section{4.4 Header Files}

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

\texttt{vxbIntrCtrlr.h}

The file \texttt{vxbIntrCtrlr.h} contains information needed for retrieving interrupt routing information from the BSP. Include this file as follows:

\begin{verbatim}
#include <hwif/vxbus/vxbIntrCtrlr.h>
\end{verbatim}

\texttt{vxbIntCtlrLib.h}

Interrupt controller drivers should also include \texttt{vxbIntCtlrLib.h} when they use \texttt{vxbIntCtlrLib} routines, which is strongly recommended. This header file is located in:

\begin{verbatim}
installDir/vxworks-6.x/target/src/hwif/intCtrl
\end{verbatim}

Therefore, Wind River interrupt controller drivers simply include it using quotation marks.

\begin{verbatim}
#include "vxbIntCtlrLib.h"
\end{verbatim}

When a third-party interrupt controller driver is released, the driver should be located in the following directory:

\begin{verbatim}
installDir/vxworks-6.x/target/3rdparty/vendor/driver
\end{verbatim}

In order to include \texttt{vxbIntCtlrLib.h}, the makefile in this directory should be modified to add \texttt{-I$(TGT_DIR)/src/hwif/intCtrl} to the \texttt{EXTRA_INCLUDE} macro as follows:

\begin{verbatim}
EXTRA_INCLUDE=-I$(TGT_DIR)/h -I$(TGT_DIR)/src/hwif/intCtrl
\end{verbatim}

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

\begin{verbatim}
#include <vxbIntCtlrLib.h>
\end{verbatim}
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 intrCtrlInputs, which is defined in the following file:

`installDir/vxworks.6.x/target/h/hwif/vxbus/vxbIntrCtrlr.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.

```c
/*
 * intrCtrlInputs 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
 * intrCtrlInputs 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, vboxIntCtlrLib 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.

```c
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:

```c
    { 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.52) 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/vxbIntrCtrlr.h
```

The structure is defined as follows:

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

```c
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.49.

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
```

The is defined as follows:

```c
/*
 * 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.

```c
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.

```c
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_INTCTRL_ISR_CALL()`
- `VXB_INTCTRL_PINENTRY_ENABLED()`
- `VXB_INTCTRL_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 the following file:

`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.55.

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 `vxbDynIntConnect()`. 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 **intCplrTableArgGet()**

*intCplrTableArgGet()* 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 **intCplrTableFlagsGet()**

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

4.6.9 **intCplrTableIsrGet()**

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

4.6.10 **intCplrHwConfShow()**

*intCplrHwConfShow()* 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
*intCplrHwConfShow()* to provide output related to *isrHandle*.

4.6.11 **intCplrTableCreate()**

*intCplrTableCreate()* 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 **intCplrTableFlagsSet()**

*intCplrTableFlagsSet()* 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 *vxbIntCplrLib.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 **intCtrlTableUserSet()**

`intCtrlTableUserSet()` 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 `vxbIntCtrlLib` 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 `vxbIntCtrlLib` 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 needs to know how many ISRs are connected to an interrupt input, the driver can call `intCtrlTableIsrGet()`. 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.

```c
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()`.

```c
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:

\[
\text{hwPrio}(\text{swPrio}(N)) \leq \text{hwPrio}(\text{swPrio}(N-1))
\]

Table 4-1 shows the possible mappings between hardware priority and software priority for 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>
<thead>
<tr>
<th>Hardware Priority</th>
<th>Software Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( N ) through ( N+3 )</td>
</tr>
<tr>
<td>1</td>
<td>( N+4 ) through ( N+7 )</td>
</tr>
<tr>
<td>2</td>
<td>( N+8 ) through ( N+11 )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>31</td>
<td>( N+256 ) through ( 0xffffffff )</td>
</tr>
</tbody>
</table>

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.
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, 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 `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:

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

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.
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:

```c
/* 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.42.

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.54.

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 \texttt{intCtrlPinFind()}. It must then check 
the value returned by \texttt{intCtrlPinFind()} to see if the value is \texttt{VXB_INTR_DYNAMIC}. 
If so, the routine follows the same procedure it does when the \texttt{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 \textit{Programming Dynamic 
Vectors}, p.54.

\begin{quote}
\textbf{NOTE:} The interrupt controller is responsible for programming the dynamic 
 vectors into the device.
\end{quote}

\section*{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 \texttt{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 \texttt{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 \texttt{vxbIntDynaVecProgram()} driver method and call it, by simply calling through 
the function pointer \_\_func\_vxbIntDynaVecProgram:

\begin{verbatim}
if ( \_\_func\_vxbIntDynaVecProgram != NULL )
{
  (*\_\_func\_vxbIntDynaVecProgram)(devID, dynaVec);
}
\end{verbatim}

The prototypes for the driver method and \_\_func\_vxbIntDynaVecProgram are as 
follows:

\begin{verbatim}
STATUS \_\_func\_vxbIntDynaVecProgram
|
  VXR\_DEVICE\_ID pInst,
  struct vxbIntDynaVecInfo *dynaVec
}
\end{verbatim}

\section*{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 \texttt{vxbIntCtlrLib} functionality, the interrupt inputs must be represented by the structures used by \texttt{vxbIntCtlrLib}. The data are kept in a structure, referred to as the \texttt{isrHandle}, which contains information about all interrupt inputs and the ISRs that are connected to them.

The information kept in the \texttt{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 \texttt{VXB_INTCTLRLIB_LOWLVL\_SIZE} and \texttt{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 \texttt{VXB\_INTCTLRLIB\_TOPLVL\_SIZE} and \texttt{VXB\_INTCTLRLIB\_LOWLVL\_SIZE} defined in the following:

\begin{verbatim}
installDir/vxworks-6.x/target/src/hwif/intCtlr/vxbIntCtlrLib.h
\end{verbatim}

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:

\[
\text{(} \text{VXB\_INTCTLRLIB\_TOPLVL\_SIZE * VXB\_INTCTLRLIB\_LOWLVL\_SIZE} \text{)}.
\]

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.

```c
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_INTCTRL_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.

The pinCpu field is configured using the hardware resource configuration in the
BSP directory (hwconf.c). Use the intrCtlrCpu data structure—which is used only
in SMP systems—to configure an interrupt to be routed to a specific CPU. For
example:

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

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
library2.

### 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.

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.

---

2. For additional information about the use of isrDeferLib in an SMP environment, see
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
```

The structure is defined as follows:

```c
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_DISCONNECT_FUNC ipiDisconnectFunc; /* 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;
```

```c
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:

```c
/***************************************************************************/
/* ipiGen - Generate Inter Processor Interrupt */
/* This function 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 iipiID
values ranging from zero to seven.

Depending on the system configuration, one or more iipiID values are reserved for
system use. The remainder may be available for application use. iipiID 1 is always
reserved for debug support, and is not available to applications. When the optional
VxWorks SMP product is used, iipiID 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.
5.1 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:

```c
#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.65).
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:

```c
VXB_DEVICE_ID vxbDevStructAlloc()
```

This routine allocates a device structure.

**vxbDeviceAnnounce()**

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

```c
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:

```c
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:

```c
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:

```c
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:

<table>
<thead>
<tr>
<th>Base</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>pRegBase[0]</td>
<td>base+0x00000000</td>
<td>0x0000ffe0 / 65504</td>
</tr>
<tr>
<td>pRegBase[1]</td>
<td>base+0x000207e0</td>
<td>0x00000020 / 32</td>
</tr>
<tr>
<td>pRegBase[2]</td>
<td>base+0x00010040</td>
<td>0x000000c0 / 192</td>
</tr>
<tr>
<td>pRegBase[3]</td>
<td>base+0x00020914</td>
<td>0x00000004 / 4</td>
</tr>
<tr>
<td>pRegBase[4]</td>
<td>base+0x00023ff0</td>
<td>0x00000010 / 16</td>
</tr>
</tbody>
</table>

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:

```c
    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 \texttt{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 \textit{VxWorks Device Driver Developer’s Guide (Vol. 1): Development Strategies}. 
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, including IPNET-native network 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 in this documentation.

**NOTE:** For IPNET-native MAC drivers and some older VxBus MAC drivers, the term END2 (IPNET-native) or END is used in the source code.
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 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](image)

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, serial network transports, and wireless Ethernet drivers. 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.

---

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.
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 multiplexer (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 decouple 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.

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.

IPNET-Native Drivers

VxWorks 6.7 introduces a new network (MAC) driver type called IPNET-native drivers. IPNET-native drivers are a form of VxBus network driver that is designed to work with the IPNET-native packet structures used by the Wind River
Network Stack. This is in contrast to traditional VxBus-enabled network drivers which use M_BLK/C_BLK/cluster tuples. Using a driver with native support for the IPNET-native packet structures improves performance in certain situations by eliminating the need for the network stack to translate between Ipcom_pkt structures and the traditional M_BLK/C_BLK/cluster tuples used by standard VxBus network drivers.

**NOTE:** This chapter applies to both IPNET-native and traditional M_BLK-based drivers. Differences between the two driver styles are noted where applicable. For more information on migrating from traditional M_BLK-style drivers to IPNET-native drivers, see VxWorks Device Driver Developer's Guide (Vol. 3): Migrating to IPNET-Native 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.

An IPNET-native driver allocates network buffers from a central pool—and provides them to the receive DMA engine of each device it manages—to accept frames that are received in the future. When a device delivers a received frame into one of these buffers, the driver replaces that buffer with a new one, and calls the MUX-provided receive routine to deliver the buffer with the frame to the appropriate network protocol. When the protocol code is done with the packet, it frees the buffer, returning it to the central pool. (When the IPNET network stack is configured in the image, the central pool is the IPNET packet pool.)

For M_BLK-style VxBus MAC drivers, each instance of the 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. After having been loaned to the network stack, there is no guarantee that any individual buffer will be returned to the driver in a particular order or in a timely manner.
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 `mux2send()`, `muxTkSend()`, or `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 driver’s interrupt handling code 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 event 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 Handling Method*, p. 110.
6.2.2 VxBus Driver Methods for Network Interface Drivers

Traditional M_BLK-style MAC drivers are required to support the \{muxDevConnect\}() driver method. IPNET-native MAC drivers are required to support the alternate \{mux2DevConnect\}() driver method. These two methods behave in the same manner, and only one can be implemented for any given driver. However, you should note that the \{mux2DevConnect\}() method is called later in the initialization sequence, after the IPNET packet pool is set up.

Network interface drivers are also likely to use several other methods, including: \{vxbDrvUnlink\}(), \{miiMediaUpdate\}(), \{miiRead\}(), and \{miiWrite\}().

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.130).

\{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 (managed by drivers that provide the method) during the network stack initialization.

The following is an example from the ns83902VxbEnd driver:

```c
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 **nicEndLoad()** to the load routine defined in the driver.
- Changing the references of **pDrvCtrl->nicMuxCookie** 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.

```c
{muxDevConnect2}()
```

The **{muxDevConnect2}()** driver method provides the mechanism for binding the network device to the network stack. This method is invoked on every MAC instance managed by drivers providing the method. The method is invoked during the network stack initialization, after the IPNET packet pool—from which IPNET-native drivers allocate packet buffers—is created.

The following is an example from the **vxbGei825xxEnd2** driver:

```c
/***************************************************************************/
/*
* geiMuxConnect - 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 deferred
* until the networking subsystem is ready. This routine will ultimately
* trigger a call to geiEndLoad() to create the END interface instance.
* 
* RETURNS: N/A
* 
* ERRNO: N/A
*/

LOCAL void geiMuxConnect
{
    VXB_DEVICE_ID pDev,
    void * unused

    GEI_DRV_CTRL *pDrvCtrl;
    pDrvCtrl = pDev->pDrvCtrl;

    /* Create our MII bus. */
    if (pDrvCtrl->geiDevType == GEI_DEVTYPE_TOLAPAI)
        {
            pDrvCtrl->geiMiiDev = vxbInstByNameFind("intelGcu", 0);
            pDrvCtrl->geiMiiPhysRead = vxbDevMethodGet(pDrvCtrl->geiMiiDev,
                                                    (UINT32)&miiRead_desc);
            pDrvCtrl->geiMiiPhysWrite = vxbDevMethodGet(pDrvCtrl->geiMiiDev,
                                                     (UINT32)&miiWrite_desc);
        }

    miiBusCreate (pDev, &pDrvCtrl->geiMiiBus);
    miiBusMediaListGet (pDrvCtrl->geiMiiBus, &pDrvCtrl->geiMediaList);
    miiBusModeSet (pDrvCtrl->geiMiiBus,
                   pDrvCtrl->geiMediaList->endMediaListDefault);
```
/* Save the cookie. */

pDrvCtrl->geiMuxDevCookie = muxDevLoad (pDev->unitNumber,
    geiEndLoad, "", TRUE, pDev);

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

if (_func_m2PollStatsIfPoll != NULL)
    endPollStatsInit (pDrvCtrl->geiMuxDevCookie,
        _func_m2PollStatsIfPoll);

return;
}

This version from the vxbGei825xxEnd2 driver is slightly atypical because a more
generic VxBus network driver creates its MII bus earlier in its devInstanceInit2() function. The gei driver delays creation of the logical MII bus until the
{mux2DevConnect}() method is called to ensure that the intelGcu logical
device—required to support the MDIO interface on the EP80579 Integrated
Processor—has been created. In addition to this driver, several other network
device drivers delay MII logical bus creation until their {muxDevConnect}() or
{mux2DevConnect}() method is called for similar reasons (for example, if PHYs
for multiple devices must all be managed through a single device’s interface).

In addition to loading the device into the MUX and starting the device, a driver
that supports polled-mode statistics collection would enable that support in this
method routine. All IPNET-native drivers should support polled-mode statistics
collection when the device hardware provides packet statistics registers.

NOTE: To support the MIB2 interface library (m2IfLib), polled-mode statistics
collection is used even if the statistics must be collected in software by the driver
on a per-packet basis. Because the alternate endM2Packet() interface—a
per-packet function call—is M_BLK oriented, it is not appropriate for IPNET-native
drivers.

{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.
The \texttt{miiMediaUpdate}() method has the same signature as the following stub routine:

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

This method is invoked by the MII bus layer whenever \texttt{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. Normally, the \texttt{miiMonitor} task checks for link events every two seconds.

The \texttt{pInst} argument is a pointer to the MAC driver’s instance handle. The MAC driver can use this 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 \texttt{miiBusModeGet}() routine.

The \texttt{miiMediaUpdate}() method is also typically used by drivers to notify bound protocols of link state changes. You must determine the new link state. If the device instance is administratively up (that is, not stopped), you must announce the change to any bound protocols. You can notify the protocols by posting a job to the driver’s job queue for the device, to execute either \texttt{muxLinkUpNotify}() or \texttt{muxLinkDownNotify}(). The code that does this should be similar to the following example from the etsec driver:

```c
/* If status went from down to up, announce link up. */
if (pDrvCtrl->etsecCurStatus & IFM_ACTIVE && !(oldStatus & IFM_ACTIVE))
    jobQueueStdPost (pDrvCtrl->etsecJobQueue, NET_TASK_QJOB_PRI,
                    muxLinkUpNotify, &pDrvCtrl->etsecEndObj,
                    NULL, NULL, NULL, NULL);

/* If status went from up to down, announce link down. */
else if (!(pDrvCtrl->etsecCurStatus & IFM_ACTIVE) && oldStatus & IFM_ACTIVE)
    jobQueueStdPost (pDrvCtrl->etsecJobQueue, NET_TASK_QJOB_PRI,
                    muxLinkDownNotify, &pDrvCtrl->etsecEndObj,
                    NULL, NULL, NULL, NULL);
```

In addition, this method should update MIB information (the RFC 1213 \texttt{ifSpeed} member and the RFC 2233 \texttt{ifHighSpeed} member) so that the correct interface speed values are reported to SNMP queries.

---

2. In Wind River provided drivers, the routine that implements the \texttt{miiMediaUpdate}() method typically uses a naming convention like \texttt{devLinkUpdate} where \texttt{dev} is a short abbreviation for the device type. For example, \texttt{geiLinkUpdate} for the \texttt{gei} driver.
Assuming the variable `speed` has the current interface speed in bits per second, and `pEnd` points to `END_OBJ`, the following code can accomplish this in any recent VxWorks version:\(^3\):

```c
pEnd->mib2Tbl.ifSpeed = speed;
if (pEnd->pMib2Tbl != NULL)
{
    pEnd->pMib2Tbl->m2Data.mibIfTbl.ifSpeed = speed;
    pEnd->pMib2Tbl->m2Data.mibXIfTbl.ifHighSpeed =
        (speed + 500000) / 1000000;
}
```

The `miiRead()` driver method allows PHYs on the MII bus to access your device’s MII management registers. The `miiRead()` method has the same signature as the following stub routine:

```c
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`.

The `miiWrite()` driver method allows PHYs on the MII bus to access the device MII management registers. The `miiWrite()` method has the same signature as the following stub routine:

---

3. The test of `pMib2Tbl` and the assignment to one of the `ifSpeed` members is redundant in VxWorks 6.7 and later.
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 `miiWrite()` routine should only return `OK` if the register is successfully updated.

### 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**
  
  For all VxBus MAC drivers:
  
  ```
  #DEFINE END_MACROS
  #include <endLib.h>
  #include <endMedia.h>
  #include <etherMultiLib.h>
  #include <netLib.h>
  ```

  For IPNET-native VxBus drivers only, the following are included in addition to those listed above:
  
  ```
  #define IPCOM_SKIP_NATIVE_SOCK_API
  #include <ipcom_vxworks.h>
  #include <ipcom_clib.h>
  #include <vxmux_pkt.h>
  #include <ipnet_eth.h>
  ```

- **MIB2 Interface**
  
  ```
  #include <m2IfLib.h>
  #include <m2Lib.h>
  ```

- **VxBus Utilities**
  
  ```
  #include <hwif/vxbus/vxBus.h>
  #include <hwif/vxbus/hxConf.h> /* for PLB devices */
  #include <hwif/vxbus/vxPciLib.h> /* for PCI devices */
  #include <hwif/util/vxdmaBufLib.h>
  #include <hwif/util/vxbParamSys.h>
  #include <../src/hwif/h/mii/miiBus.h>
  #include <../src/hwif/h/vxbus/vxbAccess.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`, `miifName`, and `miifUnit`. 
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 phyAddrN, where N is a number indicating where in the sequence the device is connected.

For more information, see 6.3 PHY Drivers, p.127.

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:

```c
VXBDEVICE_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:

```c
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) (This is for M_BLK-style VxBus drivers only, for IPNET-native drivers this is managed by the IPNET packet pool.)
- DMA support (vxbDmaBufLib)

These libraries are discussed further in the following sections.
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. Called by a VxBus MAC driver’s **muxDevConnect()** or **mux2DevConnect()** routine.

**muxDevStart()**

Starts a device already loaded into the MUX. Called by a VxBus MAC driver’s **muxDevConnect()** or **mux2DevConnect()** routine. This routine also allocates packets from the driver’s pool (**M_BLK**-style drivers) or from the global pool (IPNET native drivers), enables device interrupts, and enables a device for reception and transmission of packets.

**muxioctl()**

Accesses control routines.
muxDevStop()
Stops a previously started device. This routine can be called from a driver’s vxbDrvUnlink() routine, during muxDevUnload(), or separately. This routine quiesces the device, disabling interrupts, reception, and transmission. It also returns packet buffer resources that are held by the device or driver to the pool from which they came.

muxDevUnload()
Unloads a device from the MUX. This routine is called from the VxBus driver’s vxbUnlink() routine. Note that the device should be stopped prior to calling muxDevUnload(). However, if the device is still up, this routine attempts to stop it.

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. (For more information, see 6.2.8 jobQueueLib: Deferring ISRs, p.91).

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

```c
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 parameter rxQueue00, if specified by the BSP or application, is used to find the queue in place of netJobQueueId. The value of the rxQueue00 parameter is a pointer to a structure of type HEND_RX_QUEUE_PARAM.

**NOTE:** In this release of VxWorks, the implementation of the Wind River Network Stack specifies that only task tNet0 can execute much of the protocol code. In particular, only tNet0 can execute the code paths for packets received from a driver. This implies that if the stack might bind to a given network device, the driver for that device must queue any work for the device to the network job queue serviced by tNet0. The queue ID for that queue is netJobQueueId. However, this restriction does not apply to devices to which only non-IPNET protocols will bind. Because of this, the restriction must be imposed at the system configuration level, and network drivers must not attempt to enforce it, other than by using netJobQueueId as the default job queue when another queue is not specified by rxQueue00.

The jobQueueStdPost() routine can be used instead of jobQueuePost() when additional flexibility is required. jobQueueStdPost() does not require
pre-allocating a QJOB structure. However, when possible, use jobQueuePost() rather than jobQueueStdPost() for performance-critical code. For more information, see the reference entry for jobQueueStdPost().

If there is a requirement for a custom job queue, the jobQueueCreate() or 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

Buffer management for M_BLK-style drivers is different from that required for IPNET native drivers. Both driver types are covered in this section.

IPNET-Native Drivers

Unlike traditional M_BLK-oriented MAC drivers, IPNET-native drivers allocate packet memory out of a shared (non-device-specific) packet pool. When the network stack is included in the VxWorks image, this packet pool is the IPNET packet pool; besides being shared by all IPNET-native devices, the pool is used by socket-level kernel code that allocates packet buffers for application data to be sent.

IPNET-native drivers allocate packet buffers as follows:

```c
Ipcom_pkt * pkt;
...
pkt = vxipcom_pkt_malloc (pDrvCtrl->etsecMaxMtu, 0);
```

or sometimes:

```c
pkt = vxipcom_pkt_malloc (pDrvCtrl->etsecMaxMtu, IPCOM_FLAG_FC_EXACT);
```

Here, `pDrvCtrl->etsecMaxMtu` indicates (for the etsec driver) the MTU of the maximum sized packet supported by the link; for an Ethernet driver, this value is typically 1500, or 9000 if jumbo frames are supported. The MTU must be specified because the packet pool contains buffers of more than one size.

The buffers returned by vxipcom_pkt_malloc() are larger than the specified MTU, and include additional space for packet headers, including the maximum size link header; as well as additional alignment padding of at least one cache line after the nominal end of the buffer. Moreover, the buffers are guaranteed to start on addresses that have at least cache line alignment.

Drivers for devices with buffer alignment requirements that are stricter than cache line alignment, or drivers that, for any reason, require extra buffer space (at a given nominal MTU) beyond what is provided by the IP stack by default, can request such stricter alignment or minimum per-buffer space beyond the nominal MTU by calling the routine end2BufferPoolConfig(), provided by the vxmux_end2 module.

```c
void  end2BufferPoolConfig
{
    alignment,  /* minimum alignment of pkt->data */
    extra_space /* minimum buffer space beyond nominal MTU */
};
```

The routine has an effect only if called before the IPNET packet pool is created. Therefore, IPNET-native drivers that call this routine typically do so at the end of their devInstanceInit2() routine. A driver can pass zero for either the alignment
or the extra_space argument to avoid increasing the requirements for that argument.

If vxipcom_pkt_malloc() is called with a first argument of 0, it returns a pointer to an Ipcom_pkt structure with no attached buffer. However, this feature is not typically useful to an IPNET-native driver.

The second argument to vxipcom_pkt_malloc() is a context argument indicating whether the call is made from a special context, if it can block, or if the packet should be allocated from a memory cache determined strictly by the first (MTU) argument to vxipcom_pkt_malloc(). Generally, the driver should pass either zero (0) or IPCOM_FLAG_FC_EXACT for this second argument. If the driver wants to guarantee that the packet buffer comes from the same memory cache any time it allocates a packet specifying a fixed MTU, the driver must use the special value IPCOM_FLAG_FC_EXACT. Specifying this value means that when the driver allocates two packets, pkt1 and pkt2, specifying the same MTU, the cookie values pkt1->data_freefunc_cookie and pkt2->data_freefunc_cookie that record what memory cache each packet data buffer is allocated from are the same. This allows the driver to separate Ipcom_pkt structures from the underlying data buffers and reassociate them with different data buffers, knowing that all the buffers have the same size and come from the same memory cache so that the reassigned packets are freed correctly. This is discussed further in Receive Handling Method, p. 110.

Without the IPCOM_FLAG_FC_EXACT flag, the vxipcom_pkt_malloc() routine can (in rare instances) return a packet from memory cache containing larger than required buffers if the cache containing the best-fit buffers is found empty.

A driver must check that the value returned from vxipcom_pkt_malloc() is not IP_NULL. If the return value is IP_NULL, this indicates that the packet pool is exhausted. Because the packet pool grows automatically, up to a configurable global allocation limit across all IPNET memory pools, an IP_NULL return should rarely occur under normal operation; nevertheless, drivers must check for it.

The driver must free a packet buffer in one of two ways. A packet, pkt, that was passed to the driver send routine should be freed as follows:

VXIPCOM_PKT_DONE (pkt);

For example, you can use this method to free a packet from the transmit cleanup routine, or to free a packet left in the transmit ring when the device is stopped.

Any other (non-transmit path) packet should be freed using the following:

vxipcom_pkt_free (pkt);

For IPNET-native drivers and the network stack, packets are represented by the Ipcom_pkt data type. This type is defined in the following header file:

installDir/components/ip_net2-6.xx/ipcom/include/ipcom_pkt.h

However, IPNET-native network drivers should include this header indirectly using the following code:

#include <vxmux_pkt.h>

Ipcom_pkt is the packet header control structure that the network stack uses to refer to a packet; it contains a pointer to the actual buffer containing the packet data, several integer byte offsets into the buffer, the length of the full data buffer, and many other members (elided here):

typedef struct Ipcom_pkt_struct
{
  ...

The member `data` points to the start of the buffer associated with the `Ipcom_pkt` structure. The buffer is guaranteed to start at an address that is at least cache-line aligned, although the driver can assume greater alignment if it requests it with a call to `end2BufferPoolConfig()` from its `devInstanceInit2()` routine. The usable length of the buffer is given by the `maxlen` member. For packets allocated together with a data buffer, `data_freefunc_cookie` is an opaque value that indicates to the implementation what underlying memory cache the data buffer came from and should be returned to.\(^4\)

The members `start`, `end`, `ipstart`, and `tlstart` are byte offsets from `pkt->data`. `pkt->start` is the offset in the buffer at which packet data starts. `pkt->end` is the offset in the buffer at which packet data ends; so `pkt->end` minus `pkt->start` is the length of the packet. `pkt->ipstart` and `pkt->tlstart` are the offsets in the buffer at which the IP header and transport layer header, respectively, start. Generally, only drivers that support checksum offload (or non-Ethernet drivers that need to provide a special `hdrParse()` routine) need to be concerned with `ipstart` and `tlstart`.

Typically, the network stack passes only single-segment packets to the driver for transmission. However, if `zbuf` sockets are in use, or if an `M_BLK`-oriented protocol sends multi-segment `M_BLK` packets over an IPNET-native driver, the driver's send routine may be passed packets consisting of more than one segment of data. In this case, the separate segments making up the packet are chained through the `next_part` member. The last segment has its `next_part` member set to `IP_NULL`.

When `vxpcom_pkt_free()` or `VXIPCOM_PKT_DONE()` is passed the first `Ipcom_pkt` in a multi-segment chain, it frees the entire chain. A driver must deliver received packets to the MUX as single-segment packets; multi-segment packets are not supported in this case.

Because the packet pool is shared between devices, an IPNET-native driver should take care to return all of the packet buffers it holds for a device to the pool when the device is stopped.

Note that packets held in the transmit ring should be freed using `VXIPCOM_PKT_DONE()`, while packets in the receive transfer ring should be freed using `vxpcom_pkt_free()`. Or, if the receive transfer ring contains just data buffers

---

\(^4\) The `data_freefunc_cookie` member is also used as a general argument to a `data_freefunc` function pointer member, omitted in the example above, that indicates how an external buffer allocated separately from a bare `Ipcom_pkt` structure should be freed. However, IPNET-native drivers do not typically follow this usage.
separated from Ipcom_pkt buffers rather than full Ipcom_pkt packets, with
vxpcom_pktbuf_free( ) as follows:

vxpcom_pktbuf_free (pBuffer, data_freefunc_cookie);

where pBuffer points to the first byte of the buffer and data_freefunc_cookie is the
saved data_freefunc_cookie member from the Ipcom_pkt with which the buffer
was originally allocated. Using the IPCOM_FLAG_FC_EXACT flag when allocating
the buffers avoids having to remember a separate data_freefunc_cookie value for
each buffer.

M_BLK-Style Drivers

The routines in netBufLib are used to manage a pool of buffers, along with
buffer-specific information contained in structures known as M_BLK and C_BLK.
The M_BLK and C_BLK describe the packet data, and are the structures used by
the network stack. The M_BLK, C_BLK, 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 M_BLK, C_BLK, 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. This support is available for both M_BLK-style and IPNET-native network drivers. These routines are described in VxWorks Device Driver Guide (Vol. 1): Device Driver Fundamentals.

To support IPNET-native drivers, the library vxbDmaEnd2BufLib extends vxbDmaBufLib to provide a routine to load a DMA buffer map for a packet described by an Ipcom_pkt (or a chain of Ipcom_pkt structures in the multi-segment case). The routine vxbDmaBufMapIpcomLoad() is used in exactly the same way as vxbDmaBufMapMblkLoad(), except that the packet is described by a pointer to an Ipcom_pkt rather than to an M_BLK. The prototype for vxbDmaBufMapIpcomLoad() is as follows:

```c
STATUS vxbDmaBufMapIpcomLoad
   (VXB_DEVICE_ID pInst,
    VXB_DMA_TAG_ID dmaTagID,
    VXB_DMA_MAP_ID map,
    Ipcom_pkt * pkt,
    int flags)
```

### 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:

```c
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:

```c
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:

```c
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.130. 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 \( \text{miiRead}(\) \) method to access the PHY.

**miiBusModeSet()**

The \text{miiBusModeSet()} routine is defined as follows:

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

Like \text{miiBusModeGet()} routine, 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 \text{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 \text{miiBusModeGet()} and \text{miiBusModeSet()} as infrequently as possible. For example, your MAC driver could call \text{miiBusModeGet()} only when its \{miiMediaUpdate\} method is invoked and then cache the results, rather than calling \text{miiBusModeGet()} every time its \text{EIOCGIFMEDIA ioctl()} handler is called.

**miiBusMediaListGet()**

The \text{miiBusMediaListGet()} routine is defined as follows:

```c
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 \text{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 \text{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 for M_BLK-style drivers (see \text{muxDevConnect()}, p.74), or the \{mux2DevConnect\} method for IPNET-native drivers (see \text{muxDevConnect2()}, p.75). The actual initialization occurs from the network initialization code. For more information, see 6.2.7 MUX: Connecting to Networking Code, p.90.

When the network stack initialization code calls the driver’s \text{func muxDevConnect()} routine, the routine calls \text{muxDevLoad()}, specifying the \text{endLoad()} entry point into the driver. The \text{muxDevLoad()} routine calls the
specified `endLoad()` entry point two times. The first time the `endLoad()` entry is called, the argument contains a pointer to a buffer containing an empty string (that is, the first byte of the buffer is zero). The driver must write the driver name stem (for example `fei`, for the `fei8255xVxBEnd` driver) into the buffer 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 calls the `EIOCGSTYLE` and (possibly) `EIOCGNPT` network driver `ioctl()` routines to determine the driver style. IPNET-native network drivers must support the `EIOCGSTYLE` network driver `ioctl()` and must pass back the style `END_STYLE_END2` through the `ioctl` argument. Traditional M_BLK oriented network drivers do not need to support `EIOCGSTYLE` or `EIOCGNPT`; they are assumed to have the style `END_STYLE_END`. Note that an OK return from the `EIOCGNPT` identifies an NPT-style driver; this driver style is not supported in VxWorks 6.7 or subsequent releases.

Using the determined driver style, the MUX completes the `END_OBJ` structure by setting various structure members, including a function pointer that the driver receive handler calls to deliver received packets 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 interface directly with network layer protocols. Rather, it 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](image)
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 driver. However, this document refers to such code simply as “the protocol.”

A driver’s `muxDevConnect()` or `mux2DevConnect()` method must be called from outside the normal VxBus initialization phases. This is because a device cannot be loaded into the MUX and started until the MUX itself is initialized, as well as the packet pool support required to start the device. However, the MUX and the network packet pool support are initialized during startup of the network stack, which is done after normal VxBus device initialization is complete. When the stack has been initialized sufficiently to load and start network devices, the registered `muxDevConnect()` or `mux2DevConnect()` methods for VxBus device instances are called, causing those devices to be loaded into the MUX and started. Network stack initialization can then continue. Note that the `mux2DevConnect()` calls occur somewhat later than the `muxDevConnect()` calls, if IPNET is in the image.

Legacy network device initialization using `endDevTbl[]` occurs just after the `muxDevConnect()` method calls, but before the `mux2DevConnect()` calls.

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 perform a significant amount of work at interrupt level. The device ISR in these drivers typically drains any packets in the receive ring, at least queueing them for the stack, but sometimes also executing a fair amount of network stack code. The ISR also typically frees any completed packets in the transmit ring.

Because VxWorks is intended for real-time applications, ISRs must be kept short. Wind River does not recommend the use of long ISRs for network packet processing. For this reason, most of the network stack processing for incoming packets and completed transmits that would—in some operating systems—be done from within an ISR, is pushed to a task context in VxWorks. This is accomplished using `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 acknowledge the event causing the interrupt, and when appropriate, disable further device interrupts of the same nature. To minimize the time that the current interrupt delays task-level execution (or servicing of other, lower priority interrupts), the ISR should handle only those tasks that require minimal execution time. 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.82.)

Note: You can also use `jobQueuePost()` to queue up work other than processing of received packets.

Caution: The `jobQueuePost()` routine requires that you provide a pre-allocated QJOB structure describing the job being posted. This QJOB is typically embedded in the driver control structure for the device instance. A given enqueued QJOB may not be reused until it has been taken off the job queue to be serviced; if the same QJOB is posted again to a job queue while it is currently enqueued, the second `jobQueuePost()` call fails with no effect.

6.2.9 Working with Ipcom_pkt Packets

IPNET-native network drivers allocate packet buffers out of a shared global pool. When the IPNET stack is configured into the VxWorks image—as is usually the case when IPNET-native drivers are used—the global pool is the IPNET packet pool. This pool also supplies packet buffers used by socket-level kernel code to hold application data sent on sockets.

You can build a VxWorks image that does not include the IPNET stack, yet still supports IPNET-native drivers. To do this, edit the following file to define `VXMUX_USE_PKT_POOL_MIN`:

```
installDir/components/ip_net2-6.x/vxmux/config/vxmux_config.h
```

Next, rebuild the `vxmux` module as well as the IPNET-native drivers. In this case, an alternate, simplified implementation of a minimal Ipcom_pkt packet pool that does not depend upon the IPNET stack is used. However, this is a rare situation, and the possibility is not discussed further in this guide, except to note that the header file `vxmux_pkt.h` defines preprocessor macros, including `vxipcom_pkt_malloc`, `vxipcom_pkt_free`, and `VXIPCOM_PKT_DONE`, whose expansion depends upon whether the full IPNET packet pool, or the minimal `vxmux` replacement packet pool, provides packets. These macros are used by IPNET-native drivers to allocate or free packets.

The IPNET network stack and IPNET-native drivers represent packets using the Ipcom_pkt structure. This is a control structure that does not contain any packet data itself, but points to a packet buffer that does. The global packet pool can provide packet buffers of a few different sizes. In VxWorks 6.7, the pool is based upon several underlying slab cache pools, one for the Ipcom_pkt control structure, and one each for a small number of different data buffer sizes. The slab cache pools can grow independently and the growth is only bounded by a configurable allocation limit applied across all memory pools.

An IPNET-native driver typically allocates packets only of a single size, large enough to hold the largest size frame received from or sent to the link. For Ethernet drivers, the link MTU is 1500 bytes, unless jumbo frames are supported, in which case the link MTU is conventionally taken to be 9000 bytes. A VxBus instance parameter `jumboEnable` is used to control whether jumbo frames are supported. An Ethernet driver supporting jumbo frames uses code similar to the following example from `tsecEndLoad()` in `vxbTsecEnd2.c` to determine the MTU in use.
/* paramDesc {
 * The jumboEnable parameter specifies whether
 * this instance should support jumbo frames.
 * The default is false. }
 */

r = vxbInstParamByNameGet (pDev, "jumboEnable", VXB_PARAM_INT32, &val);
if (r != OK || val.int32Val == 0)
    pDrvCtrl->tsecMaxMtu = TSEC_MTU;
else
    pDrvCtrl->tsecMaxMtu = TSEC_JUMBO_MTU;

The driver header file defines TSEC_MTU as 1500, and TSEC_JUMBO_MTU as 9000. The tsec driver allocates packets using vxipcom_pkt_malloc() as follows:

Ipcom_pkt * pkt;
...
pkt = vxipcom_pkt_malloc(pDrvCtrl->tsecMaxMtu, 0);

The MTU does not include the size of the link-level header, and does not in any way account for packet alignment. In fact, the packet buffers returned by vxipcom_pkt_malloc() are bigger than just the specified MTU. Note the following regarding these buffers:

- The buffers contain enough extra space for the largest link header in the system (a calculation that considers not just supported hardware link types like Ethernet, but also the possibility of supported tunnelling protocols).
- The buffers include an additional amount of space, IPNET_PKT_ALIGNMENT (default value is 64), nominally for alignment.

The above items increase a packet buffer's actual usable length (pkt->maxlen) beyond the requested MTU. However, the packet buffers are actually even bigger than that. That is:

- The packet buffers are guaranteed (in VxWorks) to start on a cache-line boundary. The previously mentioned nominal alignment space (IPNET_PKT_ALIGNMENT) is not consumed to produce this alignment.
- The packet buffer size is increased by another cache-line size beyond pkt->maxlen, perhaps to cope with certain devices that might DMA up to a cache line beyond the nominal end of the buffer.

For many network drivers, the cache-line alignment guarantee for packet buffers is sufficient. After allocating a packet, an Ethernet driver for a device with minimal alignment requirements for receive transfer buffers can set the starting byte offset within the packet buffer as follows:

pkt->start = ((pkt->maxlen - pDrvCtrl->myMaxMtu) & ~3) - 18);

In this case, 18 is a nominal maximum Ethernet header size, allowing a VLAN tag as part of the header. This method leaves as much space as possible for adding tunnelling headers in front of the received packet, and guarantees that the IP header starts on a 4-byte boundary.

NOTE: The IPNET stack can handle packets with IP headers that start on an arbitrary 2-byte boundary on any architecture supported by VxWorks.

However, some devices impose a stricter alignment requirement on the starting address of receive transfer buffers. For instance, the tsec driver requires that receive buffers, into which it will write packet data, start on a 64-byte boundary.
The `tsec` device is used on several boards where the cache line is only 32 bytes, which means that the alignment requirement for the start of packet data (64 bytes) is stricter than the 32-byte alignment guarantee for the start of the packet data buffer (that is, `pkt->data`). The IPNET-native `tsec` driver must take this into account when initializing the starting offset in the receive packet buffers that it allocates. For example, the following is the code in `tsecEndStart()` that allocates and initializes packet buffers to fill the receive ring, and also initializes receive DMA buffer descriptors:

```c
/* Set up the RX ring. */
for (i = 0; i < TSEC_RX_DESC_CNT; i++)
{
    pkt = vxipcom_pkt_malloc (pDrvCtrl->tsecMaxMtu, 0);
    if (pkt == NULL)
    {
        END2_TX_SEM_GIVE (pEnd);
        semGive (pDrvCtrl->tsecDevSem);
        return (ERROR);
    }
    /*
     * `off' is the smallest offset from `pkt->data' that has 64-byte
     * alignment.
     */
    off = (- (UINT32)pkt->data) & (TSEC_RX_ALIGN - 1);
    pkt->start = off +
        ((pkt->maxlen - off - pDrvCtrl->tsecMaxMtu - MAX_ETHER_LINKHDR) &
        ~(TSEC_RX_ALIGN - 1));
    pDrvCtrl->tsecRxPkt[i] = pkt;
    pDesc = &pDrvCtrl->tsecRxDesc[i];
    pDesc->bdAddr = (UINT32)&pkt->data[pkt->start];
    pDesc->bdLen = 0;
    pDesc->bdSts = TSEC_RBD_E|TSEC_RBD_I;
    if (i == (TSEC_RX_DESC_CNT - 1))
        pDesc->bdSts |= TSEC_RBD_W;
}
```

This code positions the packet as far to the rear of the packet buffer as possible, to leave as much space as possible for inserting additional headers; but still guarantees 64-byte alignment of the packet start. Note that `pkt->data` is where the packet buffer starts, while `&pkt->data[pkt->start]` or equivalently `pkt->data + pkt->start`, is the address within the packet buffer where packet data starts. The `tsec` driver uses similar code to adjust `pkt->start` when allocating replacement receive buffers in its receive handler routine.

---

5. As the packet moves up and down the stack, IPNET may adjust `pkt->start` to point to the start of the data belonging to the current protocol layer, but `pkt->data` stays fixed.
Supporting Scatter-Gather with IPNET-Native Drivers

Scatter-gather is a DMA technique that allows for a single large block of data to be distributed among multiple buffers. When data is being sent, the DMA engine within the device gathers data from each buffer 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 buffers, 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. For reception, the IPNET stack requires that packets delivered to it be described as a single, unsegmented packet buffer. Therefore, the device’s receive DMA engine should be programmed in this way.

The network stack normally passes packets to be transmitted within a single, unsegmented packet buffer. However, if zero copy (zbuf) sockets are in use, the network stack can send multi-segment packets to the driver send routine for transmission. Also, some non-IPNET network protocols send multi-segment packets for transmission over devices managed by IPNET-native drivers. Therefore, an IPNET-native driver’s send routine must be able to handle multi-segment packets. Such packets are passed as a chain of *Ipcom_pkt* structures linked through the *next_part* member, and terminated by a *next_part* member of *IP_NULL*.

When a device does not support gather-transmit and the driver's send routine is passed a multi-segment packet, the driver must coalesce the packet into a single buffer. This involves allocating a new single packet buffer (large enough for the link MTU) from the packet pool, and copying the packet data segments from the original *Ipcom_pkt* chain into the newly allocated packet buffer. 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 address and size pairs. A driver written for such a device walks the *Ipcom_pkt* chain, extracts the data segment addresses and lengths, and then forms a gather list according to the device’s specification. This is typically faster than the copy operation required when the *Ipcom_pkt* chain is coalesced.

Each segment, or link of the *Ipcom_pkt* 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 chain, or if there are not enough scatter-gather slots in the descriptor, then the packet cannot be sent using scatter-gather, but must be coalesced into a single packet buffer. The driver send routine is responsible for determining whether the *Ipcom_pkt* 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 describe the packet data, a send routine must count the number of segments in the chain\(^6\) and then compare that number with the number of available descriptors or the number of scatter-gather slots available in a descriptor. If the chain cannot be sent using scatter-gather, the driver must either coalesce the chain into a single buffer allocated from the packet pool, or it must return *IP_ERRNO_EWOULDBLOCK* for the original packet, so that the caller maintains ownership, and can choose to free the packet or send it again later after a transmit restart notification.

---

\(^6\) As part of what it does, `vxbDmaBufMapIpcomLoad()` counts the number of segments in the chain, setting the result in the *nFrags* member of the specified `VXB_DMA_MAP`. 
When coalescing a segmented transmit packet, a driver that supports checksum offload or VLAN tag insertion must take care not to lose the checksum offload information present in the first segment of the original packet. Generally, the driver copies the relevant fields from the original first Ipcom_pkt pkt to the coalesced packet pTmp, as follows (assuming pTmp->start has been set earlier in the coalescing code):

\[
\begin{align*}
pTmp->flags &= pkt->flags; \\
pTmp->chk &= pkt->chk; \\
pTmp->ipstart &= pTmp->start + (pkt->ipstart - pkt->start); \\
pTmp->tlstart &= pTmp->start + (pkt->tlstart - pkt->start); \\
pTmp->link_cookie &= pkt->link_cookie;
\end{align*}
\]

### 6.2.10 netBufLib: Transferring Data with M_BLKs

Network drivers pass received packet data to the MUX and are passed packet data to transmit from the MUX. The data are kept in structures called M_BLKs. The routines in netBufLib provide a means of managing the M_BLK structures and the data they contain.

Each network interface requires its own memory pool for data and for M_BLK structures. 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 M_BLK structures, so they can be returned to the system pool.

> **NOTE:** The network stack can pass zero-length M_BLK chains. If the device your driver supports attempts to transmit zero-length buffers, your driver must ensure that zero-length data packets are not transmitted.

The term *cluster* is used to refer to buffers containing packet data.

#### Setting Up a Memory Pool for M_BLK-Style Drivers

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. 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 mvYukonIVxbEnd 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.

```c
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, 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 \texttt{endPoolTupleGet( )}.

\begin{alltt}
    pMblk = endPoolTupleGet (pDrvCtrl->ynEndObj.pNetPool);
\end{alltt}

\section*{Supporting Scatter-Gather with M\_BLK-Style Drivers}

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 M\_BLK chain. The network stack copies the data into the chained clusters, then sends the fragmented packet to the driver as an M\_BLK chain.

When scatter-gather is not supported and the network stack sends a fragmented packet to the driver, the driver must coalesce the M\_BLK chain. This involves allocating a cluster from the driver pool and copying the packet fragments from the M\_BLK 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 M\_BLK 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 M\_BLK chain is coalesced.

Each fragment, or link of the M\_BLK 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 M\_BLK chain, or if there are not enough scatter-gather slots in the descriptor, then the M\_BLK 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 M\_BLK 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 M\_BLK
chain, and compare that number with the number of available descriptors or the number of scatter-gather slots available in a descriptor. If the M_BLK chain cannot be sent using scatter-gather, the driver must either coalesce the M_BLK 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.

### 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.

**Checksum Offload for Internet Family Protocols**

TCP/IP checksum offload eliminates host-side checksum calculation and verification overhead by performing checksum computation with hardware assist. Many devices provide support for this feature.

The Wind River network stack and network drivers can jointly support offloading transport-layer checksum calculations for TCP and UDP over both IPv4 and IPv6, for both transmitted and received packets. Offload of VLAN tag insertion and removal can also be supported.\(^7\)

The programming interface for checksum offload between the network stack and network devices consists of two parts:

- A network driver `ioctl()`-based interface that allows a driver to report the offload capabilities of the device it manages, and allows the stack to—at a coarse level—administratively enable or disable certain of those capabilities.
- A per-packet interface for the stack to request checksum offload on packets to be transmitted, or for the driver to report checksum offload results on received packets.

The network driver `ioctl()`-based interface for reporting or managing offload capabilities is mostly the same between traditional M_BLK-oriented MAC drivers and IPNET-native drivers; but the per-packet offload interface is significantly different.

A network driver that supports checksum offload (or certain VLAN-related capabilities) must implement the `EIOCGIFCAP` and `EIOCSIFCAP ioctl()` commands. The first is called by the stack to get the (offload related) capabilities of the interface, while the second can be called to administratively enable or disable particular capabilities. Both of these `ioctl()` routines provide a pointer to an `END_CAPABILITIES` structure as the `ioctl()` argument. This structure is defined in:

```
installDir/vxworks-6.x/target/h/endCommon.h
```

---

\(^7\) Previous versions of the Wind River Network Stack also supported checksum offload for IPv4 headers. However, due to the marginal benefit that such checksum offload gives, the current stack does not attempt to support offload of the IPv4 header checksum, and provides no interface to request it or report it on a per-packet basis.
The structure is defined as follows:

typedef struct _END_CAPABILITIES
{
    uint32_t cap_available; /* supported capabilities (RO) */
    uint32_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;

For the EIOCGIFCAP command, the driver must fill in all fields of the structure. It sets cap_available to a bit mask with bits on indicating the capabilities it supports, taken from the following:

IFCAP_RXCSUM
   The device can perform some checksum offload on received packets.

IFCAP_TXCSUM
   The device can perform some checksum offload on transmitted packets.

IFCAP_JUMBO_MTU
   The device supports an MTU of 9000 bytes.

IFCAP_VLAN_MTU
   The device supports a “VLAN-compatible” MTU. That is, the addition of a 4-byte VLAN tag to the MAC header does not require decreasing the MTU by 4 bytes (for example, from 1500 to 1496).

IFCAP_VLAN_HWTAGGING
   The device supports insertion of VLAN tags per-packet on transmission in hardware, or supports extraction of VLAN tags from received packets.

In addition, the following flags are defined in the interface, but are not fully supported at present by the Wind River Network Stack:

IFCAP_TCPSEG
   The device supports TCP segmentation of “large TCP sends” in hardware.

IFCAP_IPSEC
   The device supports IPSEC cryptographic offload.

IFCAP_NETCONS
   The device can act as a network console.

IFCAP_IPCOMP
   The device can do IPcomp.

IFCAP_CAP0
   Vendor-specific capability 0.

IFCAP_CAP1
   Vendor-specific capability 1.

IFCAP_CAP2
   Vendor-specific capability 2.

IFCAP_CAP3
   Vendor-specific capability 3.

The cap_enabled member is set with the currently enabled subset of the available capabilities. At load time, the convention is that the driver sets cap_enabled equal
to `cap_available`. However, the stack may subsequently change `cap_enabled` using the `EIOCSIFCAP ioctl()` routine.

The `csum_flags_tx` and `csum_flags_rx` members give a more detailed description of the driver’s capabilities, using a different set of flags. The `csum_flags_tx` member is important because it governs what sorts of per-packet offload requests the stack can make of the driver on transmit. The `csum_flags_rx` member is currently only informational. The `csum_flags_tx` and `csum_flags_rx` members use the same set of flags that are used as `M_BLK` flags as part of the per-packet checksum offload interface for `M_BLK`-style drivers. These flags are not used in the per-packet interface for IPNET-native drivers. However, they are still used as part of the capabilities interface. The flags are defined in:

```
installDir/vxworks-6.x/target/h/wrn/coreip/net/mbuf.h
```

Table 6-1 shows provides the descriptions for these flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Transmit/Receive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSUM_IP a</td>
<td>TX</td>
<td>The device can calculate and insert the IPv4 header checksum.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can verify the IPv4 header checksum.</td>
</tr>
<tr>
<td>CSUM_TCP</td>
<td>TX</td>
<td>The device can calculate and insert the TCP checksum in a TCP packet over IPv4.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can verify or calculate the TCP checksum of a TCP packet over IPv4.</td>
</tr>
<tr>
<td>CSUM_UDP</td>
<td>TX</td>
<td>The device can calculate and insert the UDP checksum in a UDP packet over IPv4.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can verify or calculate the UDP checksum of a UDP packet over IPv4.</td>
</tr>
<tr>
<td>CSUM_TCPv6</td>
<td>TX</td>
<td>The device can calculate and insert the TCP checksum in a TCP packet over IPv6.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can verify or calculate the TCP checksum of a TCP packet over IPv6.</td>
</tr>
<tr>
<td>CSUM_UDPv6</td>
<td>TX</td>
<td>The device can calculate and insert the UDP checksum in a UDP packet over IPv6.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can verify or calculate the UDP checksum of a UDP packet over IPv6.</td>
</tr>
<tr>
<td>CSUM_VLAN</td>
<td>TX</td>
<td>The device can insert an arbitrary VLAN tag into the Ethernet header of an outgoing frame, on a per-packet basis.</td>
</tr>
<tr>
<td></td>
<td>RX</td>
<td>The device can extract and remove the VLAN tag from a received packet and report it to the stack.</td>
</tr>
</tbody>
</table>
The following capabilities flags are defined but are not supported by the stack for VxWorks network drivers:

**CSUM_IP_FRAGS (TX)**
The device can perform checksum across IPv4 fragments.

**CSUM_FRAGMENT (TX)**
The device can perform IPv4 fragmentation of frames too large for the link MTU.

**CSUM_TCPSEG (TX)**
The device can perform TCP segmentation of large TCP/IPv4 sends.

**CSUM_TCPv6_SEG (TX)**
The device can perform TCP segmentation of large TCP/IPv6 sends.

The following flags are not used as capabilities, but rather are used only in the per-packet offload interface for M_BLK oriented drivers, in the M_BLK mFlags member. They are not used at all for IPNET-native drivers, but are listed here for completeness:

**CSUM_IP_CHECKED (RX)**
The IPv4 header checksum for this packet was checked.

**CSUM_IP_VALID (RX)**
The IPv4 header checksum for this packet was found to be valid. Used only in combination with CSUM_IP_CHECKED.

**CSUM_DATA_VALID (RX)**
The csum_data field in the M_PKT_HDR structure is valid. Contents depends on whether or not CSUM_PSEUDO_HDR is also set.

**CSUM_PSEUDO_HDR (RX)**
If set, the csum_data field contains the un-complemented ones’ complement sum over the TCP or UDP header, transport payload, and the IP pseudo-header used in calculating the TCP or UDP checksum. If not set, the sum in the csum_data field does not include the IP pseudo-header sum. This flag is significant only in conjunction with CSUM_DATA_VALID.

The following are simply combinations or aliases of various previously listed flags, for convenience:

- CSUM_DELAY_DATA is (CSUM_TCP | CSUM_UDP)
- CSUM_DELAY_IP is (CSUM_IP)
- CSUM_DELAY_DATA6 is (CSUM_TCPv6 | CSUM_UDPv6)
The following macros are used only in the M_BLK-oriented model to extract transmit checksum-offload related information from a packet. They do not apply to IPNET-native drivers, which do not use M_BLK structures:

**CSUM_IPHDR_OFFSET**

Extracts the byte offset of the start of the IP header from the start of the packet.

```c
#define CSUM_IPHDR_OFFSET(pMblk) ((pMblk)->mBlkHdr.offset1)
```

**CSUM_IPHDR_LEN**

Extracts the length in bytes of the IP header. For IPv6, this includes any extension headers before the transport header.

```c
#define CSUM_IP_HDRLEN(pMblk) ((pMblk)->mBlkHdr.offset2)
```

**CSUM_XPORT_HDRLEN**

Extracts the length in bytes of the transport-layer header. Note that this macro is not supported by the network stack.

```c
#define CSUM_XPORT_HDRLEN(pMblk) 
  (((pMblk)->mBlkPktHdr.csum_data & 0xff00) >> 8)
```

**CSUM_XPORT_CSUM_OFF**

Extracts the byte offset of the transport layer checksum field relative to the start of the transport header:

```c
#define CSUM_XPORT_CSUM_OFF(pMblk) ((pMblk)->mBlkPktHdr.csum_data & 0xff)
```

The network stack (or a management application) can use the `EIOCSIFCAP ioctl()` to change which offload capabilities of the driver are enabled. The caller sets the `cap_enabled` member of the `END_CAPABILITIES` structure specified to the desired set of capabilities. The driver `ioctl()` code sets its current capabilities accordingly, ignoring any capabilities that the caller specified but the driver does not support. The driver can, but usually need not, modify the device hardware settings in response to a change in enabled capabilities. For instance, if the device supports receive checksum offload, but a management utility calls the `EIOCSIFCAP ioctl()` to disable `IFCAP_RXSUM`, current drivers can discard the checksum offload information reported by the device for received datagrams, rather than passing it on to the stack; receive checksum offload is still enabled in the hardware.

A network driver’s `EIOCGIFCAP` and `EIOCSIFCAP ioctl()` handlers are usually simple, as in this fragment of a switch statement from the `vxbEtsecEnd2.c` driver’s `etsecEndIoctl()` routine:

```c
case EIOCGIFCAP:
  hwCaps = (END_CAPABILITIES *)data;
  if (hwCaps == NULL)
    {
      error = EINVAL;
      break;
    }
  hwCaps->csum_flags_tx = pDrvCtrl->etsecCaps.csum_flags_tx;
  hwCaps->csum_flags_rx = pDrvCtrl->etsecCaps.csum_flags_rx;
  hwCaps->cap_available = pDrvCtrl->etsecCaps.cap_available;
  hwCaps->cap_enabled = pDrvCtrl->etsecCaps.cap_enabled;
  break;
```
case EIOCSIFCAP:
    hwCaps = (END_CAPABILITIES *)data;
    if (hwCaps == NULL)
        { 
            error = EINVAL;
            break;
        }
    pDrvCtrl->etsecCaps.cap_enabled = hwCaps->cap_enabled;
    break;

Per-Packet Transmit Checksum Offload Interface (IPNET-Native Drivers)

This section discusses the per-packet checksum offload interface used with IPNET-native drivers. To request transport-layer checksum calculation and insertion on a packet to be transmitted, IPNET sets the IPCOM_PKT_FLAG_HW_CHECKSUM bit in the flags member of the lead Ipcom_pkt for the packet. The stack can request transport checksum offload for UDP, TCP, or ICMP over IPv4 or IPv6. Note the following:

- The stack will only request such transmit checksum offload on a datagram if the driver had set IFCAP_TXCSUM in the cap_available member of the END_CAPABILITIES structure passed to EIOCGIFCAP. For IPv4 datagrams, the driver must also have set both CSUM_TCP and CSUM_UDP in the csum_flags_tx member; and for IPv6 datagrams, the driver must have set both CSUM_TCPv6 and CSUM_UDPv6 in csum_flags_tx.

- It is the responsibility of the stack to not request transmit checksum offload when it has administratively disabled IFCAP_TXCSUM using EIOCSIFCAP.

- The stack does not request transmit checksum offload on IP fragments.

When requesting transmit checksum offload on a packet pkt, the stack provides certain information for use by the driver as follows:

- pkt->chk contains the byte offset of the two byte transport checksum field within the transport-layer header. This value is 6 for UDP, 16 for TCP, and 2 for ICMP (IPv4 or IPv6). The driver can use this value to infer the transport protocol, as well as to locate the transport-level checksum field.

- The transport checksum field in the packet contains, in network byte order, the uncomplemented ones-complement sum over the IP pseudo-header used in the transport checksum calculation. This pseudo-header covers the IP source address, IP destination address, a two byte transport-layer length field, and a two byte field in which the first byte is zero and the second byte is the number identifying the transport protocol from the IPv4 protocol field or the IPv6 next header field. Most devices either expect the transport checksum field to be initialized this way, or do not care about the value in the checksum field because they calculate the pseudo-header checksum themselves and ignore the transport checksum field during the hardware checksum calculation.

- pkt->start contains the byte offset of the start of the packet (link header) within the packet data buffer.

- pkt->ipstart contains the byte offset of the start of the IP header within the packet data buffer. Therefore, pkt->ipstart minus pkt->start is the offset from the start of the packet to the start of the IP header. If the driver needs to determine whether the network layer protocol is IPv4 or IPv6, it can examine the byte pkt->data[pkt->ipstart]; the high order four bits of this byte contains the IP version number, for both IPv4 and IPv6.
- **pkt->tlstart** contains the byte offset of the start of the transport layer header within the packet data buffer. Therefore, **pkt->tlstart** minus **pkt->ipstart** is the size of the IP header, including any extension headers or options preceding the transport header.

- If the stack requests transmit checksum offload on a multi-segment packet, the link header, network layer header, and transport layer header are guaranteed to occur in the first segment of the packet.

If for any reason, the device cannot service the transmit checksum offload request for the given packet, the driver must recognize this and must calculate (and set) the checksum in software. It does this by calling the following routine:

```c
vxmux_calculate_transport_checksum(pkt);
```

For example, although the IPNET stack requests ICMP checksum offload if the driver advertises itself capable of UDP and TCP checksum offload, the device may not in fact be capable of calculating and inserting the ICMP header checksum. In this case, the driver must calculate the checksum in software. As another example, some devices may be able to compute TCP or UDP checksums, but only when there are no IPv4 options or IPv6 extension headers present. A driver for a device with such limitations must either not claim to be able to do the transmit checksum offload, or must catch the exceptional case packets and compute the checksum for them in software.

If a driver advertises the **IFCAP_VLAN_HWTAGGING** capability, and if it sets **CSUM_VLAN** in the csum_flags_tx member of the END_CAPABILITIES structure in EIOCGIFCAP, and if the stack is configured to support VLANs, then the stack may request that the device insert a particular 802.1q VLAN tag into the (Ethernet) link header. It makes this request by setting the bit **IPCOM_PKT_FLAG_VLAN_TAG** in **pkt->flags**, and providing the desired 16-bit tag control information (TCI) word. The TCI word may be extracted from the packet using the following code:

```c
vtag = IPNET_ETH_PKT_GET_VLAN_TAG(pkt);
```

Here **vtag** is returned in network byte order. The VLAN tag header in the MAC header is 32 bits, composed of a fixed Tag Protocol Identifier value (0x8100) followed by the 16-bit TCI field. Having extracted the TCI, the driver programs the device to insert the tag header.

Below is example code from the `geiEndAdvEncap()` routine in the `vxbGei825xxEnd2` driver. This code, which supports an Intel 82575 device, initializes an advanced format TCP/IP context descriptor in the transmit DMA ring. This sets up the device with packet format information to perform checksum offload or VLAN tag insertion, when flagged to do so by data descriptors later in the transmit ring. The context set up by a context descriptor is kept in hardware and affects subsequently transmitted packets until another context descriptor is set. The 82575 actually provides 16 on-chip contexts, but the driver code shown below makes use of only one of them. The code only writes a new context descriptor when offload is requested and the packet layout or the VLAN TCI differs from that set for the previous context descriptor.

The particular device details are not the focus of this example, but the example does illustrate how a driver can make use of the information passed when the network stack requests transmit checksum offload or VLAN tag insertion for a packet.
/*
 * Set up a TCP/IP offload context descriptor if needed. Note,
 * we must handle VLAN tag insertion here also.
 */

if (IP_BIT_ISSET (pkt->flags, (IPCOM_PKT_FLAG_HW_CHECKSUM |
                    IPCOM_PKT_FLAG_VLAN_TAG)))
{
    UINT32 v1 = pkt->ipstart - pkt->start;
    UINT32 v2 = pkt->tlstart - pkt->ipstart;
    UINT32 offsets = (pkt->chk + (v1 << 8) + (v2 << 16));
    vtag = IPNET_ETH_PKT_GET_VLAN_TAG(pkt);

    if (offsets != pDrvCtrl->geiLastOffsets ||
        (IP_BIT_ISSET (pkt->flags, IPCOM_PKT_FLAG_VLAN_TAG) &&
         vtag != pDrvCtrl->geiLastVlan))
    {
        GEI_STATS (txCtxDesc++);

        pDrvCtrl->geiLastOffsets = offsets;
        pDrvCtrl->geiLastVlan = vtag;

        pCtx = (GEI_ADV_CDESC *)&pDrvCtrl->geiTxDescMem[pDrvCtrl->geiTxProd];
        val = htole32(GEI_ADV_TDESC_CMD_DEXT | GEI_ADV_TDESC_DTYP_CTX);

        /*
         * We identify UDP by a transport header checksum offset of 6.
         * If not UDP (e.g. for TCP, ICMP) there's no special handling of
         * the case of a zero checksum value; for UDP this case gets
         * inverted to 0xffff.
         *
         * So treat everything but UDP as TCP.
         * If we eventually support TCP segmentation offload, we may need
         * to revisit this.
         */
        if (pkt->chk != 6)
            val |= htole32(GEI_ADV_CDESC_L4T_TCP);

        /* distinguish IPv4 from IPv6 */
        if ((pkt->data[pkt->ipstart] & 0xf0) == 0x40)
            val |= htole32(GEI_ADV_CDESC_CMD_IPV4);

        pCtx->gei_cmd = val;

        val = GEI_ADV_MACLEN(v1) | GEI_ADV_IPLEN(v2);
        pCtx->gei_macip = htole16(val);

        if (IP_BIT_ISSET (pkt->flags, IPCOM_PKT_FLAG_VLAN_TAG))
            pCtx->gei_vlan = htole16(vtag);

        pDrvCtrl->geiTxFree--;
        GEI_INC_DESC(pDrvCtrl->geiTxProd, GEI_TX_DESC_CNT);
    }
}

Per-Packet Receive Checksum Offload Interface (IPNET-Native Drivers)

If a device managed by an IPNET-native driver receives an IP datagram on which
the device performs transport-layer checksum verification or calculation—and if
the receive checksum offload capability (IFCAP_RXCSUM) has not been
administratively disabled—the driver can report the checksum results to the
network stack by setting certain flags and fields in the Ipcom_pkt structure
describing the received packet. Alternatively, if IFCAP_RXCSUM has been
administratively disabled, the driver must not provide checksum offload information in the Ipcom_pkt it delivers to the MUX.
Devices that support receive checksum offload for TCP or UDP usually provide a **pass**, **fail**, or **not-tested** indication for the transport checksum; or else provide the actual ones-complement sum over the transport header and payload. If the actual sum is provided, it may or may not include the IP pseudo-header sum that is part of the transport layer checksum definition. In some cases, the device may provide just a raw sum that covers more of the packet than the transport header and payload (and the pseudo-header).

If the device provides a simple “pass” or “fail” indication for the transport level checksum (on packets for which it tests the checksum at all) then the driver should do the following:

- If the device reports that the UDP, TCP, or ICMP layer checksum is correct, the driver should set the bit `IPCOM_PKT_FLAG_HW_CHECKSUM` in `pkt->flags`, and set `pkt->chk` to 0xffff as follows:

```c
pkt->chk = 0xffff;
IP_BIT_SET(pkt->flags, IPCOM_PKT_FLAG_HW_CHECKSUM);
```

Note that 0xffff is the expected uncomplemented ones-complement sum over the IP pseudo-header, transport header, and transport payload, for a correct transport-layer checksum.

- If the device reports that it did not test the checksum on the particular received packet, or that the checksum was tested but found to be incorrect, the driver should **not** set `IPCOM_PKT_FLAG_HW_CHECKSUM` in `pkt->flags` and it should **not** write to `pkt->chk`. In these cases, the stack checks the checksum in software. (Note that we are trusting the device to tell us when the checksum is correct, but are not trusting it to tell us when it is wrong.)

If the device does not provide a pass-fail indication for the transport layer checksum, but does provide the ones-complement sum over either the IP pseudo-header, the transport layer header, the transport payload, and no other part of the packet; or just the transport layer header and payload, then the driver should do the following:

- If the packet is UDP and the value in the received UDP header’s checksum field is zero, that indicates that the sender did not supply the UDP checksum. The driver should not provide checksum offload information in this exceptional case; the stack handles it in software.

Otherwise:

- Store the device-calculated 16-bit uncomplemented ones-complement sum in `pkt->chk` in network byte order.

- In `pkt->flags`, if the device-provided checksum includes the IP pseudo header, set the bit `IPCOM_PKT_FLAG_HW_CHECKSUM`. If the device-provided checksum does **not** include the pseudo-header, set the bit `IPCOM_PKT_FLAG_TL_CHECKSUM`.

If the device provides only the ones-complement sum over a portion of the frame that is different from either just the transport layer header and payload, or the transport layer header and payload plus the IP pseudo-header, then the driver must do one of the following:

- Adjust the checksum in software to cover only the transport header and payload (and possibly the IP pseudo-header), and then report the checksum as in one of the cases above.

- Not report checksum offload information at all for the packet.
In deciding which choice to take, you should consider that performing transport checksum adjustments at the driver level involves work that is not valuable if the packet is not destined for the local host (something that is difficult to tell at the driver level). If the packet is only going to be forwarded, doing work in the driver related to the transport layer checksum is not productive.

The following is example code from the `vxbEtsecEnd2` driver’s `etsecEndRxHandle()` routine. This example illustrates checksum offload handling for received packets. The `etsec` device reports transport layer checksum verification status in a fixed-size frame control block (FCB) preceding the received packet data in memory. If the receive checksum offload is administratively enabled and the FCB flags indicate that the transport checksum has been verified and no error was found, the driver sets `pkt->chk` to indicate success and sets the bit `IPCOM_PKT_FLAG_HW_CHECKSUM` in `pkt->flags`.

```c
if (pDrvCtrl->etsecCaps.cap_enabled & IFCAP_RXCSUM) {
    if (pFcb->fcbFlags & ETSEC_RX_FCB_CTU &&
        !(pFcb->fcbFlags & ETSEC_RX_FCB_ETU)) {
        pkt->chk = 0xffff;
        IP_BIT_SET(pkt->flags, IPCOM_PKT_FLAG_HW_CHECKSUM);
    }
}
```

### 802.1q VLAN Tag Extraction

If the device supports extraction of 802.1q VLAN tags from received frames (and reporting of the extracted VLAN tag control information), the driver can provide the IFCAP_VLAN_HWTAGGING capability. When this capability is administratively enabled, the driver’s receive handler passes the packet’s extracted VLAN TCI to the stack in the `Ipcom_pkt` structure. The following is example code from the `vxbGei825xxEnd2` driver’s `geiEndRxHandle()` routine:

```c
/* Handle VLAN tag removal offload */
if (pDrvCtrl->geiCaps.cap_enabled & IFCAP_VLAN_HWTAGGING) {
    if (pDesc->gei_sts & GEI_RDESC_STS_VP) {
        Ip_u16 vtag = le16toh(pDesc->gei_special);
        IPNET_ETH_PKT_SET_VLAN_TAG(pkt, vtag);
        IP_BIT_SET(pkt->flags, IPCOM_PKT_FLAG_VLAN_TAG);
    }
}
```

If IFCAP_VLAN_HWTAGGING is enabled and the receive DMA descriptor for the received packet indicates that the device stripped a VLAN tag from the packet, the driver copies the VLAN tag control information from the receive descriptor to the `link_cookie` field in the packet using the macro `IPNET_ETH_PKT_SET_VLAN_TAG` (defined in `installDir/components/ipnet2-6.x/ipnet2/src/ipnet_eth.h`), and sets the flag `IPCOM_PKT_FLAG_VLAN_TAG` in `pkt->flags`.

### Checksum Offloading and Receiving (M_BLK-Style Drivers)

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.
If receive checksumming is enabled, the driver reads the device’s checksum status report for the packet 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 checksum 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 `M_BLK` 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 `M_BLK`.

**Handling Corrupt Packets (M_BLK-Style Drivers)**

Wind River recommends that if the device reports a packet to have an invalid transport checksum, the driver should treat the packet as if the device did not report any checksum status (that is, do not set `CSUM_DATA_VALID`). In this case, the stack will calculate the packet checksum in software. That is, barring known errata, trust the device to indicate when the checksum is good, but not to indicate when the checksum is bad. This allows some checksum offload benefit from devices that in certain (rare) instances, incorrectly report a packet as having a bad checksum. Of course, it is a good idea to test a driver supporting checksum offload on reception by sending it packets with a variety bad checksums as well as good ones, to validate that it can distinguish the good from the bad correctly. If a device reports some bad checksums as good, it is probably better not to enable checksum offload at all.

A driver supporting checksum offload might have code like the following in its receive handler:

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

8. The Wind River Network Stack does not make use of the `CSUM_IP_CHECKED` or `CSUM_IP_VALID` bits; it always calculates the IPv4 header checksum in software.
/ * Determine if IP checksum calculated */
if (csumStatus & RFD_CS_IP_CHECKSUM.Bit_VALID)
{
   /* Set mBlk checksum flags to indicate checksum calculated */
   pRbdTag->pMblk->m_pkthdr.csum_flags |= CSUM_IP_CHECKED;
}
/* Determine if IP checksum valid */
if (csumStatus & RFD_CS_IP_CHECKSUM.VALID)
{
   /* Set mBlk checksum flags to indicate a valid IP header */
   pRbdTag->pMblk->m_pkthdr.csum_flags |= CSUM_IP_VALID;
}
if (csumStatus & RFD_CS_TCPUDP_CHECKSUM.Bit_VALID &&
    csumStatus & RFD_CS_TCPUDP_CHECKSUM.VALID)
{
   pRbdTag->pMblk->m_pkthdr.csum_flags |=
      CSUM_DATA_VALID|CSUM_PSEUDO_HDR;
   pRbdTag->pMblk->m_pkthdr.csum_data = 0xFFFF;
}

Checksum Offloading and Transmission (M_BLK-Style Drivers)

The network stack may request that the driver send routine program the device to perform checksum calculation and insertion on an outgoing packet. The stack does this by setting certain CSUM flags in the mBlkPktHdr.csum_flags member of the lead M_BLK describing the packet.

The CSUM_IP flag requests that the driver program the device to calculate the IPv4 header checksum and insert it into the (pre-zeroed) checksum field in the IP header. However, since VxWorks 6.5, the network stack does not make use of this facility, and always sets the IPv4 header checksum field to the actual valid checksum according to RFC 791.

The stack may, however, request UDP or TCP checksum offload for an outgoing packet by setting one of the following flags in mBlkPktHdr.csum_flags:

- CSUM_TCP—stack requests checksum offload for a TCP segment over IPv4
- CSUM_UDP—stack requests checksum offload for a UDP datagram over IPv4
- CSUM_TCPv6—stack requests checksum offload for a TCP segment over IPv6
- CSUM_UDPv6—stack requests checksum offload for a UDP segment over IPv6

The stack only sets one of these bits if the driver sets the same bit in the csum_flags_tx member of the END_CAPABILITIES structure returned by the its EIOCGIFCAP ioctl() routine. The network stack does not separate the capability to do TCP checksum offload from the capability to do UDP checksum offload. Therefore, it only requests either if the driver advertises the ability to do both. However, the stack does consider TCP/UDP checksum offload over IPv6 separate from checksum offload over IPv4. If the driver only advertises CSUM_TCP and CSUM_UDP, the stack never requests checksum offload for TCP or UDP over IPv6.

When the stack requests TCP or UDP checksum offload (whether over IPv4 or IPv6), the stack initializes the checksum field in the TCP or UDP header with the ones' complement sum over the IP pseudo-header. This cannot be disabled. If necessary, the driver can overwrite the value in the checksum field if the device requires different initialization of that field. The device is expected to overwrite the checksum member with the actual checksum that it calculates.
Note that this calculation is generally done as the device DMAs the packet data from system memory to its internal transmit FIFOs. Because the checksum calculation covers the whole transport payload as well as the transport header—but the checksum is stored in the transport header—the device must wait until it has calculated the full checksum before transmitting the packet to the physical media. To avoid this additional latency affecting transmission throughput, devices that support transmit checksum offload generally have larger transmit FIFOs and can support transmitting one packet onto the wire while another is read into the transmit FIFO.

The driver send routine has available to it some information about the packet layout that some devices may require to perform transmit checksum offload. This information is mostly available through utility macros defined in the same header file (net/mbuf.h) as defines the other CSUM flags:

- **CSUM_IPHDR_OFFSET** (mBlk)—The byte offset from the start of the packet to the start of the IP header; the link header size.
- **CSUM_IP_HDRLEN** (mBlk)—The length in bytes of the IP header, including options or extension headers; the offset from the start of the IP header to the start of the UDP or TCP header.
- **CSUM_XPORT_CSUM_OFF** (mBlk)—The byte offset from the start of the transport header to the transport header checksum field; 6 for UDP, 16 for TCP.

When the stack requests transmit checksum offload on a packet, the driver is expected to program the device (in a device-specific way) to perform the checksum’s calculation and insertion into the checksum field in the outgoing packet. Some devices may have idiosyncratic restrictions which prevent them from being able to perform transmit checksum offload on certain packets; for instance, a device might only be able to perform the transport-layer checksumming if the packet has no IPv4 options in the IPv4 header. In that case, the driver must perform the checksum offload itself in software, or else not claim to support transport checksum offload for IPv4 at all. The driver may use VxWorks-provided utilities such as the in_cksum_skip() routine to help in calculating IP family checksums in software.

### 6.2.12 Other Network Interface Driver Issues

This section discusses additional issues that must be handled by your driver.

**Receive Handling Method**

VxWorks network drivers defer much of the work related to servicing interrupt conditions to code executing at task level, 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, especially when under heavy load. The driver can accomplish this by following certain guidelines for its receive (receive) ISR and for the task-level receive handler function queued from the ISR. Consider the following:

- When the device receive ISR runs, it should disable further receive interrupts from the device, and queue the task-level receive handler for execution using jobQueuePost().
The network task runs the task-level receive handler function, processing received packets until there are no more to handle. Then, the receive handler routine re-enables receive interrupts from the device, and returns. Under a heavy load of received packets, the task-level receive handler routine would then service many packets before reenabling interrupts from the device.

This simple approach needs certain refinements to deal with the following two primary complications:

- If the interrupt line used by the device is shared by other devices, the device's ISR may in fact be run even when the device has interrupts disabled.
- The network task that runs the task-level receive handler routine, most often tNet0, is shared by the network stack itself and potentially many network devices. The network task runs queued jobs one by one, finishing the current job before starting the next one. For fairness to other network devices, and to other types of network jobs such as transmit cleanup handling and protocol-level timeout processing, a driver's receive handler should limit the amount of work it does in any one network job execution queued with jobQueuePost().

Consider the problem of shared interrupt lines first. Assume you have two devices A (the network device) and B (some other device, possibly another network device) that share an interrupt line. The result of the sharing is that, whether A or B generated a particular interrupt, the ISRs of device A and device B will both be run as a result, even though one of the devices may have interrupts disabled. That is, simply disabling device interrupts cannot prevent a device ISR from running if the interrupt line is shared.

The conventional strategy for ISRs in environments where this is possible is that each ISR first reads a status register of its device to see whether there is an event pending on the device that might have caused the interrupt. If not, the ISR quickly exits; it knows that its device did not cause the interrupt. The other device's ISR presumably finds that its device did cause the ISR and handles it appropriately.

However, there are some possible problems with this approach. For example:

- Many devices show an event (for example, a received packet) even if interrupts are currently disabled for that device, and the task-level handler job is already posted.
- For some devices, reading the event status register can have the side effect of acknowledging any events found. If the task-level handler is already posted, you could lose the acknowledged events, causing them to perhaps not be handled until a subsequent event occurs, for example, a later packet arrives.
- A driver typically uses a single QJOB member embedded in its control structure for the device instance to post the receive handling job. If the QJOB is currently enqueued waiting to run, posting it a second time has no effect; but you could lose information about the particular event that elicited the post. On the other hand, if the handler has already started to run (and has not reposted itself), posting the QJOB using jobQueuePost() causes it to be queued again for execution. Before the current instance of the handler returns, it may

---

9. If the device uses a single interrupt to report not just received packets (or overruns) but also non-receive events such as transmit completions and link state changes, then the same strategy can be used, employing a single task-level handler routine that handles receive packets, transmit completions, and link state changes as needed.
reenable device interrupts if it finds no more work to do. This can result in the new handler instance running with device interrupts enabled. While this is not necessarily catastrophic, this condition could persist for a long time interval, depending upon how the ISR is written and how much load there is, leading to increased CPU utilization during that interval.

You want to prevent an ISR that runs due to interrupt line sharing from reposting the receive handler job when it is already posted or executing. This is desirable both to avoid the 'handler running with interrupts enabled' issue mentioned above, and simply to avoid unnecessary work. To achieve this, some synchronization between the ISR and the task level handler is needed (because you cannot rely upon disabling device interrupts alone). If not carefully done, such synchronization is susceptible to various race conditions, from lost events (with the typical result that some packets occasionally languish in the receive ring until a subsequent packet arrives), to permanent suspension of receive processing, or even work queue panics.

Two methods have been found to work well for most devices when synchronizing between the device ISR and the task-level handler. Both use a flag that indicates whether the task-level handler has already been posted; if the ISR runs and finds this flag already set, it does not re-post the handler job. If it finds the flag clear, it disables interrupts and posts the task-level handler. The handler job is responsible for clearing the flag and reenabling device interrupts just before it returns. The two methods differ in what constitutes the "flag", and in what code acknowledges device interrupt events.

The first method is the simpler of the two; it makes use of the device interrupt mask register (MY_IMR in the code examples below) as the flag to detect whether interrupts are already masked and the task-level handler job already posted. This method also avoids acknowledging device interrupt events in the interrupt service routine; only the task-level handler acknowledges interrupt events. For this method, the device interrupt service routine and task-level handler look similar to the following example code.

Example 6-1 Method 1 Example

This example assumes a device that uses a single ISR for all event types of interest (for example, receive packet arrival or overrun, transmit completion, link state change, and so forth), and posts a single handler job to deal with any of the events. However, you could modify the example to suit a device that uses separate receive, transmit, and link state ISRs.

```c
/* myEndInt - Interrupt Service Routine to handle device interrupts */

LOCAL void myEndInt
{
  MY_DRV_CTRL * pDrvCtrl

  VXB_DEVICE_ID pDev;
  UINT32 status;
  pDev = pDrvCtrl->myDev;
  /*
  * Read interrupt status register to check for pending device events.
  * Here we assume that this does not acknowledge any such events.
  */
  ...
```

VxWorks
Device Driver Developer's Guide, 6.8
status = CSR_READ_4(pDev, MY_ISR);

/*
 * Make sure there's really an interrupt event pending for us.
 * We assume that this device may share an interrupt line
 * with another device. If so, this function might be invoked
 * as a result of the other device asserting the interrupt, in
 * which case we might not have any work to do. MY_INTRS is
 * a mask of the interrupt events that we are interested in.
 */
if ((status & MY_INTRS) == 0)
    return;

/*
 * Check the device's interrupt mask register (MY_IMR).
 * If device interrupts are already disabled (masked),
 * we are either stopped, or have already posted a job to handle
 * previous events, and it hasn't finished running yet. Don't post
 * another job in that case, just return.
 */
if (CSR_READ_4(pDev, MY_IMR) == 0)
    return;

/*
 * Otherwise, disable device interrupts here by clearing
 * the mask register
 */
CSR_WRITE_4(pDev, MY_IMR, 0);

/*
 * This atomic flag is not used here to synchronize between the
 * ISR and the task-level handler. It merely lets the driver stop
 * routine know that the handler is pending (or running), which
 * helps with clean shutdown.
 */
(void)vxAtomicSet (&pDrvCtrl->handlerPending, TRUE);

/* post the handler job */
(void)jobQueuePost (pDrvCtrl->myJobQueue, &pDrvCtrl->myIntJob);

return;
}

/***************************************************************************
*  myEndIntHandle - task level handler job for RX/TX/Link interrupts
*  *
*/
LOCAL void myEndIntHandle
{
    void * pArg
    {
        QJOB *pJob;
        MY_DRV_CTRL *pDrvCtrl;
        VXB_DEVICE_ID pDev;
        UINT32 status;

        /*
         * Convert the QJOB argument to a pointer to the driver control
         * structure for this device instance.
         */
        pJob = pArg;
        pDrvCtrl = member_to_object (pJob, MY_DRV_CTRL, myIntJob);
        pDev = pDrvCtrl->myDev;

        /*
         * Convert the QJOB argument to a pointer to the driver control
         * structure for this device instance.
         */
        pJob = pArg;
        pDrvCtrl = member_to_object (pJob, MY_DRV_CTRL, myIntJob);
        pDev = pDrvCtrl->myDev;

/*
 * Read and acknowledge interrupts here. For this device, we
 * acknowledge the interrupts by writing the value read back to
 * the interrupt status register.
 */
status = CSR_READ_4(pDev, MY_ISR);
CSR_WRITE_4(pDev, MY_AUXISR, status);

/*
 * Check if we need to do RX handling work. We do if there
 * were RX events pending, or if on the previous execution
 * we handled the maximum number of packets per job.
 */
if ((status & MY_RXINTRS) || pDrvCtrl->moreRx)
{
pDrvCtrl->moreRx = FALSE;
    /*
     * myEndRxHandle() handles a bounded number of received packets,
     * and sets pDrvCtrl->moreRx if it has to stop because it reaches
     * the maximum number per job before running out.
     */
    myEndRxHandle (pDrvCtrl);
}

/* Do transmit cleanup work if needed. */
if (status & MY_TXINTRS)
{
    myEndTxCleanup (pDrvCtrl); /* free all completed TX resources */
}

/* Handle link state changes if necessary. */
if (status & MY_LINKINTRS)
    myLinkUpdate (pDev);

/* Check for additional interrupt events */
if (pDrvCtrl->moreRx || CSR_READ_4(pDev, MY_ISR) & MY_INTRS)
{
    jobQueuePost (pDrvCtrl->myJobQueue, &pDrvCtrl->myIntJob);
    return;
}
vxAtomicSet (&pDrvCtrl->myIntPending, FALSE);

/* re-enable interrupts here */
CSR_WRITE_4(pDev, MY_IMR, pDrvCtrl->myIntrs);
return;
}

The second method is more complex. This method uses an atomic variable to
record the interrupt events that the interrupt service routine acknowledges, as well
as to flag whether or not the task-level handler has been posted. It can potentially
be preferable to method 1 in cases where the interrupt status register cannot be
read without acknowledging events. It is also more easily adaptable to certain
hardware for which transmit and receive interrupts cannot be disabled for an
individual device.

Example 6-2  Method 2 Example

In this example all of the event bits that the driver is interested in (MY_INTRS) fit
in a 32-bit atomic variable, and there is at least one available bit left over
(MY_HANDLER_PENDING) that can be used as a flag to indicate that the task-level
handler has been posted.
/****************************************************************************
* myEndInt - Interrupt Service Routine to handle device interrupts
*/
LOCAL void myEndInt
{
    MY_DRV_CTRL * pDrvCtrl
    {
        VXB_DEVICE_ID pDev;
        UINT32 status;
        pDev = pDrvCtrl->myDev;
        /* Read the interrupt status register */
        status = CSR_READ_4(pDev, MY_ISR);
        /*
        * Make sure there’s really an interrupt event pending for us.
        * We assume that this device may share an interrupt line
        * with another device. If so, this function might be invoked
        * as a result of the other device asserting the interrupt, in
        * which case we might not have any work to do. MY_INTRS is
        * a mask of the interrupt events that we are interested in.
        */
        status &= MY_INTRS;
        if (status == 0)
            return;
        /*
        * Acknowledge interrupts by writing back the same bits.
        * For a few devices, the acknowledgement happens automatically
        * as a side-effect of the register read above, and this
        * write is not needed.
        */
        CSR_WRITE_4(pDev, MY_ISR, status);
        /*
        * Save the event bits that we acknowledged for use by
        * the task-level handler, and find out if we need to
        * post the handler job.
        */
        status = vxAtomicOr (&pDrvCtrl->eventsPending,
            status | MY_HANDLER_PENDING);
        /*
        * vxAtomicOr() returns the value of eventsPending before the
        * atomic OR operation. If the handler was already pending,
        * do nothing else. Otherwise, disable device interrupts and
        * post the job to execute the task level handler.
        */
        if (status & MY_HANDLER_PENDING)
            return;
        /* Disable device interrupts for the device. */
        CSR_WRITE_4(pDev, MY_IMR, 0);
        /* post handler job */
        jobQueuePost (pDrvCtrl->myJobQueue, &pDrvCtrl->myIntJob);
        return;
    }
}
/***************************************************************************
*  myEndIntHandle - task level handler job for RX/TX/Link interrupts
*
***************************************************************************/
LOCAL void myEndIntHandle
{
    void * pArg
}

QJOB *pJob;
MY_DRV_CTRL *pDrvCtrl;
VXB_DEVICE_ID pDev;
UINT32 status;

/*
 * Convert the QJOB argument to a pointer to the driver control
 * structure for this device instance.
 */
pJob = pArg;
pDrvCtrl = member_to_object (pJob, MY_DRV_CTRL, myIntJob);
pDev = pDrvCtrl->myDev;

/*
 * Read the current events pending saved by the ISR, and atomically
 * clear all bits in eventsPending except for MY_HANDLER_PENDING.
 */
status = vxAtomicAnd (&pDrvCtrl->eventsPending, MY_HANDLER_PENDING);

/*
 * Check if we need to do RX handling work. We do if there
 * were RX events pending, or if on the previous execution
 * we handled the maximum number of packets per job.
 */
if ((status & MY_RXINTRS) || pDrvCtrl->moreRx)
{
    pDrvCtrl->moreRx = FALSE;
    /*
     * myEndRxHandle() handles a bounded number of received packets,
     * and sets pDrvCtrl->moreRx if it has to stop because it reaches
     * the maximum number.
     */
    myEndRxHandle (pDrvCtrl);
}

/* Do transmit clean-up work if needed. */
if (status & MY_TXINTRS)
{
    myEndTxCleanup (pDrvCtrl); /* clean up all completed TX */
}

/* Handle link state changes if necessary. */
if (status & MY_LINKINTRS)
    myLinkUpdate (pDev);

/*
 * Check for additional relevant interrupt events in status
 * register MY_ISR.
 */
status = CSR_READ_4(pDev, MY_ISR) & MY_INTRS;

/*
 * Acknowledge any events by writing back the same bits.
 * For a few devices, the acknowledgement happens automatically
 * as a side-effect of the read above, and this write is not
 * needed.
 */
CSR_WRITE_4(pDev, MY_ISR, status);
Either of the two methods presented above can be applied to any number of devices. Device hardware is diverse and these methods might not work as presented in all cases. However, you should be able to use them as starting points for further development. For example, on some hardware platforms, shared interrupt lines are not possible, in which case drivers restricted to those platforms can use somewhat simpler code in the ISR(s) and task-level handler(s). Another variation is that in some cases it might be necessary to use a spinlock to protect non-atomic accesses to device registers from the ISR and the task-level handler. In this case, given that the spinlock is already used, you could use ordinary variables protected by the spinlock—rather than atomic variables—to flag that the handler is running, to save acknowledged events, and so on.

Both of the examples rely on a separate receive handler routine that limits the number of packets it delivers to the MUX in one execution (and indicates to the caller by some means if it reaches that limit).\(^\text{10}\) This helps to ensure that a single network job cannot monopolize the network task (usually `tNet0`) for too long, preventing other work that needs to be done. For example, when forwarding packets between two interfaces, doing transmit cleanup work on the egress interface with sufficient frequency is just as necessary to good performance as delivering received packets from the ingress interface. If the receive handler job for the ingress interface executes for too long, the job that does transmit cleanup for the egress interface may not get to execute, so that the egress transmit ring fills up and packets start being dropped, damaging performance.

\(^{10}\) A typical limit on the number of received packets delivered per job would be 16 or 32 packets.
In principle, the amount of transmit cleanup work done within a single network job execution ought to be bounded like the amount of packet receive work. However, transmit cleanup work is usually much faster than the packet processing done on the receive side. Also, the transmit ring is not added to while transmit cleanup is occurring, and is generally small enough that even if the full transmit ring is cleaned in a single job, it is not an excessive amount of work.

**Receive Stall Handling**

When a device receives packets, it copies each packet’s data into a buffer (cluster) provided by the driver. The device can continue to deliver received packets in this way even as the device driver code executes to process the packets and replenish the buffers. 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.

Be sure that your driver’s receive handler routine also handles the case of a receive stall in a way that is appropriate for the device.

### 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 class. 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.

Alternatively, if you build your VxWorks image using a VxWorks Image Project (VIP), you can configure the project to support the kernel shell and use a standalone symbol table. You can then add the components `INCLUDE_NET_INIT_SKIP` and `INCLUDE_WDB_COMM_SERIAL` to prevent network stack initialization from being done automatically at startup. When you are ready, you can call `sp usrNetworkInit` from the shell to start the network stack.

> **NOTE:** The default component `INCLUDE_WDB_COMM_END` is incompatible with `INCLUDE_NET_INIT_SKIP`. Adding `INCLUDE_WDB_COMM_SERIAL` is a simple way to remove `INCLUDE_WDB_COMM_END` without removing the whole WDB agent from your VIP. You can add back `INCLUDE_WDB_COMM_END` and `INCLUDE_NET_INIT_SKIP` when you are done debugging driver startup.

### 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 operating systems. 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 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.

11. The ifconfig command is also available from the command shell interpreter, where it is called without quotes around the argument list.
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()` with the `down` argument and `muxDevStop()` is the cookie returned by the `muxDevLoad()` call when you initialize your driver. You can also obtain the MUX device cookie for your driver by calling the `endFindByName()` routine. For example:

```c
-> ynCookie = endFindByName("yn", 0)
-> ifconfig ("yn0 down")
-> muxDevStop(ynCookie)
```

To restart the driver, call `muxDevStart()` and call `ifconfig()` with the `up` argument:

```c
-> muxDevStart(ynCookie)
-> ifconfig("yn0 up")
```

Driver Load and Unload

Under some conditions, you can remove a VxBus network driver from the VxBus system by calling the `vxbDrvUnregister()` routine—passing it the device driver registration ID—visible in the output from `vxBusShow()`. Among other chores, this routine calls the driver `{vxbDrvUnlink}()` method for each of the driver's device instances. As discussed previously, the `{vxbDrvUnlink}()` method is responsible for stopping the device (if necessary) by calling `muxDevStop()`, then unloading it from the MUX using `muxDevUnload()`, before freeing all resources maintained for the instance by the driver, including finally the driver control structure. During the `muxDevUnload()` call, each protocol bound to the device in the MUX has its shutdown routine called (if it provides one), and is expected to unbind from the device. `muxDevUnload()` does not return until all protocols bound to the device are unbound.

In this VxWorks release, the IPNET protocols properly unbind from a network device when the MUX calls the shutdown routine provided by IPNET for the device. However, other network services that can be bound to the device may not provide a working shutdown routine. For example, the WDB agent can bind to a network device that is used as the WDB debug channel, but the WDB agent does not provide a shutdown routine or any API to detach the WDB agent from the network interface once it is bound to the interface.

Thus, `vxbDriverUnregister()` is only supported for a network driver if the following conditions are met:

- The WDB agent is not bound to any of the network devices managed by the driver.
- For any other network service bound in the MUX to one of the driver's device instances, the service must either provide a working shutdown routine that

---

12. In VxWorks 6.5 and 6.6, the shutdown routine provided by the stack is a stub and does not cause the stack to unbind from the device. In these releases, the stack must be manually detached from the network device.
unbinds the service from the device, or else you must arrange to unbind the service before you call `vxbDriverUnregister()`.

**NOTE:** In this release, the stack detaches automatically from a network device if its shutdown routine is called. However, you can also manually detach IPNET from an interface (for example) by calling the following:

```
-> ifconfig("yn0 down")
-> ifconfig("yn0 detach")
```

This detachment method is available in VxWorks 6.5 and later releases.

There are some additional restrictions that apply to certain drivers. For instance, the `etsec` driver controls local bus Ethernet controller devices on the Freescale MPC8641D processor (and several other Freescale processors). The MPC8641D provides four separate `etsec` Ethernet controller devices, but only a single functional MDIO port that is associated with the `etsec0` device. As a result, PHY management operations for the `etsec1`, `etsec2`, and `etsec3` interfaces are delegated to the `etsec0` interface, and depend on the existence of the driver software structures representing `etsec0`. However, `vxbDriverUnregister()` unlinks the devices in the order: `etsec0`, `etsec1`, `etsec2`, `etsec3`. Certain PHY operations done when unloading the later devices can fail catastrophically because `etsec0` is already unlinked.

Several other drivers have a similar issue. The workaround for this issue is to call `vxbDeviceDriverRelease()` for each of the VxBus device instances managed by the driver—calling it for `etsec0` last—before calling `vxbDriverUnregister()` to unregister the device driver. This controls the order in which the device instances are brought down, and avoids the problem.

After a driver has been unregistered, its registration routine can be called again to reregister the driver. This recreates the device instance at the VxBus level. However, it does not load the device instances into the MUX. To load the devices, the driver's `mux2DevConnect()` or `muxDevConnect()` method must be called for each of the new device instances. Note that you should not do this by calling `vxbDevMethodRun()`, as that routine calls the method for all devices that provide it, not just for the newly created instances related to the reregistered driver. You can call `vxbDevMethodGet()` to obtain the method function pointer for one of the devices, and then call the function pointer passing it the device ID, and 0 (NULL) as a second argument.

The following is an example of unregistering and then reregistering the `gei` network driver. The example reloads its devices into the MUX, and reconnects IPNET to one of them. Originally, there are four device instances, `gei0` through `gei3`, and IPNET protocols—and no other network services—are attached to only one of them (`gei2`). `vxBusShow()` is used to find the `gei` driver registration ID. In the example, there are no special restrictions on the use of `vxbDriverUnregister()`.

```
-> muxShow
Device: gei Unit: 0 END_OBJ: 0x444c010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
Device: gei Unit: 1 END_OBJ: 0x444e010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
Device: gei Unit: 2 END_OBJ: 0x4450010 refs: 8
Description: Intel PRO/1000 VxBus END Driver
Protocol: IPSTACK IPv4 Type: 0x0800
  Recv 0x42b1fc Shutdown 0x31c3dd TxRestart 0x42b221 Arg 0x447a748
Protocol: IPSTACK ARP Type: 0x0806
  Recv 0x42b1fc Shutdown 0x31c3dd TxRestart 0x0 Arg 0x447a748
Protocol: IPSTACK RARP Type: 0x8035
```
Recv 0x42b1fc Shutdown 0x31c3dd TxRestart 0x0 Arg 0x447a748
Protocol: IPSTACK IPv6 Type: 0x86dd
Recv 0x42b1fc Shutdown 0x31c3dd TxRestart 0x0 Arg 0x447a748
Protocol: IPSTACK PPPOE DIS Type: 0x8863
Recv 0x42b1fc Shutdown 0x31c3dd TxRestart 0x0 Arg 0x447a748
Protocol: IPSTACK PPPOE SES Type: 0x8864

Device: gei Unit: 3 END_OBJ: 0x4452010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
value = 78166452 = 0x4a8b9b4
-> vxbShow

Registered Bus Types:
  MII_Bus @ 0x0046ef8c
  PCI_Bus @ 0x0046f480
  PLB_Bus @ 0x0046b364

Registered Device Drivers:
  pentiumPci at 0x0046e820 on bus PLB_Bus, funcs @ 0x0046f340
  i8253TimerDev at 0x0046f2c0 on bus MF_Bus, funcs @ 0x0046f340
  fileNVRam at 0x0046f060 on bus PLB_Bus, funcs @ 0x0046f0a0
  ns16550 at 0x0046f160 on bus PLB_Bus, funcs @ 0x0046f228
  genericPhy at 0x0046fe0 on bus MII_Bus, funcs @ 0x0046f228
  milBus at 0x0046f000 on bus PCI_Bus, funcs @ 0x0046f0a0
  milBus at 0x0046f000 on bus PLB_Bus, funcs @ 0x0046f0a0
  gei at 0x0046eb00 on bus PCI_Bus, funcs @ 0x0046eb98
  fei at 0x0046eb00 on bus PCI_Bus, funcs @ 0x0046eb98
  plbCtlr at 0x0046f060 on bus PCI_Bus, funcs @ 0x0046fa8

Busses and Devices Present:
  PLB_Bus @ 0x0047e258 with bridge @ 0x0046b3c0
  Device Instances:
    ns16550 unit 0 on PLB_Bus @ 0x0047f218 with busInfo 0x00000000
    ns16550 unit 1 on PLB_Bus @ 0x0047f418 with busInfo 0x00000000
    pentiumPci unit 0 on PLB_Bus @ 0x0047f618 with busInfo 0x0047e498
    i8253TimerDev unit 0 on PLB_Bus @ 0x00481c18 with busInfo 0x00000000
    fileNVRam unit 0 on PLB_Bus @ 0x00481d18 with busInfo 0x00000000
  Orphan Devices:
    PCI_Bus @ 0x0047e498 with bridge @ 0x0047f618
    Device Instances:
      gei unit 0 on PCI_Bus @ 0x0047fc18 with busInfo 0x00000000
      gei unit 1 on PCI_Bus @ 0x0047fd18 with busInfo 0x00000000
      gei unit 2 on PCI_Bus @ 0x00480018 with busInfo 0x00000000
      gei unit 3 on PCI_Bus @ 0x00480118 with busInfo 0x00000000
      milBus unit 0 on PCI_Bus @ 0x00481e18 with busInfo 0x00482458
      milBus unit 1 on PCI_Bus @ 0x00482018 with busInfo 0x00482518
      milBus unit 2 on PCI_Bus @ 0x004825d8 with busInfo 0x004825d8
      milBus unit 3 on PCI_Bus @ 0x00484418 with busInfo 0x00482698
  Orphan Devices:
    (null) unit 0 on PCI_Bus @ 0x0047f918 with busInfo 0x00000000
    ... some orphan device output omitted here ...
  MII_Bus @ 0x00482458 with bridge @ 0x00481e18
  Device Instances:
    genericPhy unit 0 on MII_Bus @ 0x0048f1f8 with busInfo 0x00000000
  Orphan Devices:
    MII_Bus @ 0x00482518 with bridge @ 0x00482018
    Device Instances:
      genericPhy unit 1 on MII_Bus @ 0x0048f218 with busInfo 0x00000000
    Orphan Devices:
    MII_Bus @ 0x004825d8 with bridge @ 0x00484218
    Device Instances:
      genericPhy unit 2 on MII_Bus @ 0x00484318 with busInfo 0x00000000
    Orphan Devices:
    MII_Bus @ 0x00482698 with bridge @ 0x00484418
    Device Instances:
      genericPhy unit 3 on MII_Bus @ 0x00484518 with busInfo 0x00000000

value = 1 = 0x1
-> vxbDriverUnregister 0x0046eb00
value = 0 = 0x0
muxShow value = 78166452 = 0x4a8b9b4
-> # vxBusShow would show that the gei devices have become orphans,
-> # and the HII buses (and the PHYs under them) have been removed.
-> # Reregister the gei driver.
-> geiRegister
value = 0 = 0x0
-> # vxBusShow would now show the gei devices as instances again,
-> # but the HII buses are not created for this driver until you call
-> # the muxDevConnect method. First find the device IDs. You could just
-> # look at the vxBusShow output to find them, but to do it a bit
-> # more programmatically:
-> gei = "gei"
New symbol "gei" added to kernel symbol table.
gei = 0x447afdc: value = 71779528 = 0x44744c8
-> gei0dev = vxbInstByNameFind (gei, 0)
New symbol "gei0dev" added to kernel symbol table.
gei0dev = 0x4488bac: value = 4717592 = 0x47fc18
-> gei1dev = vxbInstByNameFind (gei, 1)
New symbol "gei1dev" added to kernel symbol table.
gei1dev = 0x4452ea8: value = 4717848 = 0x47fd18
-> gei2dev = vxbInstByNameFind (gei, 2)
New symbol "gei2dev" added to kernel symbol table.
gei2dev = 0x4474528: value = 4718616 = 0x480018
-> gei3dev = vxbInstByNameFind (gei, 3)
New symbol "gei3dev" added to kernel symbol table.
gei3dev = 0x3ff6c918: value = 4718872 = 0x480118
-> muxCnx = vxbDevMethodGet (gei0dev, &muxDevConnect_desc)
New symbol "muxCnx" added to kernel symbol table.
muxCnx = 0x447c914: value = 3680675 = 0x3829a3 = geiRegister + 0x120
-> muxShow
value = 78166452 = 0x4a8b9b4
-> (*muxCnx) (gei0dev, 0)
value = 0 = 0x0
-> # This driver uses the same muxDevConnect method for all its instances...
-> (*muxCnx) (gei0dev, 0)
value = 0 = 0x0
-> (*muxCnx) (gei1dev, 0)
value = 0 = 0x0
-> (*muxCnx) (gei2dev, 0)
value = 0 = 0x0
-> (*muxCnx) (gei3dev, 0)
value = 0 = 0x0
-> muxShow
Device: gei Unit: 0 END_OBJ: 0x444e010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
Device: gei Unit: 1 END_OBJ: 0x4445010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
Device: gei Unit: 2 END_OBJ: 0x4445010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
Device: gei Unit: 3 END_OBJ: 0x444c010 refs: 2
Description: Intel PRO/1000 VxBus END Driver
value = 78166452 = 0x4a8b9b4
-> # To use gei2 again in the stack, we need to attach the
-> # stack to the interface and then configure it.
-> ipcom_drv_eth_init (gei, 2, 0)
value = 0 = 0x0
-> cmd
[vxWorks]$ ifconfig gei2 192.168.16.227/24 up
[vxWorks]$ route add default 192.168.16.1
[vxWorks]$ add net 0.0.0.0: netmask 0.0.0.0: gateway 192.168.16.1
[vxWorks]$ ifconfig -a
lo0 Link type:Local loopback Queue:none
inet 127.0.0.1 mask 255.255.255.255
inet6 unicast ::1 prefixlen 128
inet6 unicast FE80::1%lo0 prefixlen 64 automatic
UP RUNNING LOOPBACK MULTICAST
MTU:1500 metric:1 VR:0 ifindex:1
RX packets:7 mcalt:0 errors:0 dropped:4
TX packets:7 mcast:3 errors:0
collisions:0 unsupported proto:0
RX bytes:340 TX bytes:340

gei2 Link type:Ethernet HWaddr 00:04:23:a9:05:8c Queue:none

6 Network Drivers
6.2 Network Interface Drivers

capabilities: TXCSUM TX6CSUM
inet 192.168.16.227 mask 255.255.255.0 broadcast 192.168.16.255
inet6 unicast FE80::204:23FF:FEA9:58C%gni2 prefixlen 64 automatic
UP RUNNING SIMPLEX BROADCAST MULTICAST
MTU:1500 metric:1 VR:0 ifindex:3
RX packets:5 mcast:0 errors:0 dropped:0
TX packets:14 mcast:8 errors:0
collisions:0 unsupported proto:0
RX bytes:452 TX bytes:1124

[vxWorks]# ping 147.11.45.8

Pinging lx1 (147.11.45.8) with 64 bytes of data:
Reply from 147.11.45.8 bytes=64 time=0ms ttl=63
Reply from 147.11.45.8 bytes=64 time=0ms ttl=63
Reply from 147.11.45.8 bytes=64 time=0ms ttl=63
Reply from 147.11.45.8 bytes=64 time=0ms ttl=63

--- lx1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 4000 ms
rtt min/avg/max = 0/0/0 ms

[vxWorks]#

Because VxBus network drivers expect to load and unload the device into the MUX themselves, you cannot unload a VxBus network device from the MUX by calling muxDevUnload() outside of the driver's [vxbDrvUnlink()] method, and you cannot load such a device into the MUX by calling muxDevLoad() outside of the driver’s [mux2DevConnect()] or [muxDevConnect()] method.

You can use the ability to unregister then reregister a driver, to make and test changes in a driver that is built as a downloadable kernel module (DKM) rather than being linked directly in the VxWorks image. For example, you can unregister the driver. Then, from the shell, unload its module using unld(). Next, modify the driver code and rebuild the driver object module. Then, reload the new module using ld() at the shell. Note that this only works if the download can occur over some channel other than one of the devices managed by the driver being reloaded. The loading and unloading can also be done programmatically using routines from loadLib, unldLib, and moduleLib, in production applications that need to support separately loadable drivers.

However, it is not common in an actual product to unregister or reregister a device driver, or even unload its device instances. In many systems, a network driver is built into the system image and registered. Then, its devices are loaded into the MUX and started, only once when the system boots. After that point, they are left untouched. Nevertheless, drivers should be written to support dynamic registration, device unlinking, and MUX connection, as described in this section.

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.126 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.

Special Considerations for IPNET-Native Drivers

IPNET-native drivers do not make use of, or allocate, netBufLib-style M_BLK tuple pools. However, a separate component INCLUDE_END2_LINKBUFPOOL supports use of an IPNET-native network device by the WDB agent or by other M_BLK-oriented protocols that attempt to allocate M_BLK packets from a network device’s netBufLib-style pool. The INCLUDE_END2_LINKBUFPOOL component creates a single M_BLK pool (actually an M_LINK pool using the linkBufPool backend) that can be shared by all IPNET-native drivers. The INCLUDE_END2_LINKBUFPOOL component is automatically included in a VIP if the components INCLUDE_END2 and INCLUDE_WDB_COMM_END are included.

An IPNET-native driver’s load routine should have a line similar to the following (from the IPNET-native gei driver) to allow its devices to share the pool created by the INCLUDE_END2_LINKBUFPOOL:

```c
pDrvCtrl->geiEndObj.pNetPool = _end2_linkBufPool;
```

Usually, the WDB agent allocates only a small number of packets at a given time, and only from a single network device’s pool. Therefore, by default, INCLUDE_END2_LINKBUFPOOL creates a pool that is fairly small. However, the component has two parameters that can be adjusted to increase the number of tuples in the pool or the size of the associated clusters. These parameters are:

**END2_LINKBUFPOOL_NTUPLES**

The number of tuples in the IPNET-native shared linkBufPool. The default is 8.

**END2_LINKBUFPOOL_CLSIZE**

The size of tuple clusters in the IPNET-native shared linkBufPool. The default is 1600.

Note that multiple devices, possibly with different MTUs, can share the same pool. Choose a value of END2_LINKBUFPOOL_CLSIZE that is sufficient for the maximum MTU among the devices that need INCLUDE_END2_LINKBUFPOOL support. You can increase END2_LINKBUFPOOL_NTUPLES if your testing (for example, with the WTX test) reveals the need.

**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 By Example.
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 MACPHY media connection types (such as MIU, 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 the optional 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`.

Note that many boards are based on system-on-a-chip (SoC) designs with several built-in MACs and external PHYs. On these boards, the MDIO management pins of all the available PHYs are typically connected to a single set of pins on the SoC. Each PHY is strapped for a different MII bus address.

There is no reliable generic method to automatically determine on the fly which PHY address goes with which MAC. (Sometimes the PHYs are addressed in ascending order corresponding to the internal MACs, but sometimes they are addressed in descending order, and sometimes the addresses are assigned with no particular pattern based simply on how easy it is to route the PHY address strapping pins to power supply rails or to ground.)
Consequently, BSPs for such boards usually have the PHY addresses for each MAC hard coded in the BSP `hwconf.c` file. (The Ethernet driver usually has a `phyAddr` property to specify the right management address for the PHY associated with a given MAC instance.) These hard coded values are only correct for those boards with which the BSP has been tested. (For an example, see the `hwconf.c` file for your reference BSP.)

When adapting an existing BSP supplied by Wind River (or another vendor) to your own custom hardware, you must be sure to modify the PHY address configuration in `hwconf.c` to match your own board. The `phyAddr` property is used by the MAC drivers to filter accesses to all addresses except for the address assigned to a given interface instance. (Normally, `miiBus` autoprobes all PHYs on a given MII management bus, but it would be a mistake to allow it to assign multiple PHYs to the bus for a single interface, which is what would happen if the MAC driver did not take steps to prevent it.) If a driver supports a `phyAddr` property and the value in the BSP `hwconf.c` file does not match the actual board setup (for example, if an interface is configured for `phyAddr 1`, but the PHY is actually strapped for MII address 2), then the MAC driver will fail to detect a PHY device at all. This will, among other things, cause the driver to think the link is always down, and prevent it from sending any packets.

**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 `tsec` (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

The `genericPhy` driver is included with the MII bus subsystem and 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_GENERICTBIPHYPH` 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.74.

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 declared as follows:

LOCAL STATUS miiModeSet
{
    VXB_DEVICE_ID pInst,
    UINT32 mode
}
;

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 function `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 programmed 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.

The `miiModeGet()` method is declared as follows:

```c
LOCAL STATUS miiModeGet
   (VXB_DEVICE_ID pInst,
    UINT32 * mode,
    UINT32 * status)
```

The `miiModeGet()` method is used to return the current link state information. As with the `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_VALID bit if it contains valid data, and the IFM_ACTIVE bit is set if the link is up.
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:

```c
#include <../src/hwif/h/mii/miiBus.h>
```

Individual PHY drivers may also have their own header files located in the following directory:

```
installDir/vxworks-6.x/target/src/hwif/h/mii
```

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:

```c
#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.80.

**Lower Edge Utility Routines**

The following lower edge utility routines are available:

**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.
miiBusMediaAdd()
The **miiBusMediaAdd()** routine is defined as follows:

```c
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.89. A typical 10/100 Ethernet PHY might specify its media support as shown in the following example:

```c
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:

```c
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:

```c
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:

```c
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 \texttt{miiBusWrite()} routine is defined as follows:

\begin{verbatim}
STATUS miiBusWrite
{
  VXB DEVICE ID pInst,
  int phyAddr,
  int phyReg,
  UINT16 regVal
}
\end{verbatim}

This routine is used by a PHY instance to write its own registers. The \texttt{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.

\textbf{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 (\texttt{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 \texttt{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 \texttt{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 \texttt{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 \textit{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 Programmer’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, 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

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.

```c
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.
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 d1643Segments[] = {
    /* IBM Eval kit software use */
    { 0, 1024, "IBMEvalKit", 0 },
};
```
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.141 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.

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).

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).

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_J28F008` 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).

**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 following directory:

`installDir/vxworks-6.x/target/src/drv/tffs`
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.147).

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:

\[
\text{vol.erasableBlockSize} = a\text{Value} \times \text{vol.interleaving};
\]

Where \(a\text{Value}\) 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 \text{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.154). 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 \texttt{read} unchanged. Otherwise, your MTD identify routine must update this member to point to a more appropriate routine.

\textbf{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.

\textbf{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.

\textbf{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 \texttt{FLFlash} with the \texttt{powerOnCallback} member of the \texttt{FLSocket} structure. For many flash memory devices, no such routine is necessary.

\textbf{Return Value}  
The identification routine must return \texttt{flOK} or an appropriate error code defined in \texttt{flbase.h}. The stub provided is:

\begin{verbatim}
FLStatus myMTDIdentification(  
  FLFlash vol  
)  
  {  
    /* Do what is needed for identification */  
    /* If identification fails return appropriate error */  
    return flOK;  
  }
\end{verbatim}

After setting the members listed above, this routine should return \texttt{flOK}.

\textbf{Call Sequence}  
Upon success, the identification routine updates the \texttt{FLFlash} structure, which also completes the initialization of the \texttt{FLSocket} structure referenced within this \texttt{FLFlash} structure.
Writing the MTD Map Routine

MTDs need to provide a map routine only when a RAM buffer is required for windowing. No MTDs are provided for devices of this kind in this release. If the device you are using requires such support, you need to add a map routine to your MTD and assign a pointer to it in \texttt{FLFlash.map}. The routine takes three arguments, a pointer to the volume structure, a “card address”, and a length field, and returns a void pointer.

\begin{verbatim}
static void FAR0 * Map (FLFlash vol, CardAddress address, int length)
{
    /* implement function */
}
\end{verbatim}

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:

\begin{itemize}
  \item Read, write, or erase only a byte, a word, or a long word at a time.
  \item Be able to handle an unaligned read or write.
  \item Be able to handle a read, write, or erase that crosses chip boundaries.
\end{itemize}

When writing these routines, you probably want to use the MTD helper routines \texttt{flNeedVpp()}, \texttt{flDontNeedVpp()}, and \texttt{flWriteProtected()}. The interfaces for these routines are as follows:

\begin{verbatim}
FLStatus flNeedVpp(FLSocket vol)
void flDontNeedVpp(FLSocket vol)
FLBoolean flWriteProtected(FLSocket vol)
\end{verbatim}
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.  

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.

**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:

```c
static FLStatus myMTDWrite
{
    FLFlash vol,
    CardAddress address,
    const void FAR1 *buffer,
    int length,
    FLLboolean 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.

---

1. An MTD does not need to touch Vcc. TrueFFS turns Vcc on before calling an MTD routine.
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 **filesystem.h** has prototypes of compare routines that can be used for this purpose.

### Erase Routine

The erase routine must erase one or more contiguous blocks of a specified size. This routine is given a flash volume pointer, the block number of the first erasable block and the number of erasable blocks. The stub provided is:

```c
static FLStatus 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_sName**; 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 following file to include it:

```
installDir/vxworks-6.x/target/config/comps/vxworks/00tffs.cdf
```
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/flash.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`. 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.155.

### 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[]` 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 */

This table is defined in:

installDir/vxworks-6.x/target/src/drv/tffs/tffsConfig.c

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 xxxRegister() routine, the socket registration routine. This routine is responsible for assigning routines to the member functions of the socket structure.
- The routines assigned by the registration routine.
- The macro values that should reflect your hardware.

In this stub file, all of the required routines are declared. Most of these routines are defined completely, although some use generic or fictional macro values that you may need to modify.

The socket register routine in the stub file is written for RFA (resident flash array) sockets only. There is no stub version of the registration routine for PCMCIA socket drivers. If you are writing a socket driver for RFA, you can use this stub file and follow the steps described in the following section. If you are writing a PCMCIA socket driver, see the general information in Understanding Socket Driver Functionality, p.158 and the example in:

installDir/vxworks-6.x/target/src/drv/tffs/sockets/pc386-sysTffs.c

**NOTE:** Examples of other RFA socket drivers are located 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 to your BSP directory from:

```
installDir/vxworks-6.x/target/config/comps /src
```

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.156.

⚠️ CAUTION: Because you may need it for future ports, do not edit the original copy of the stub version of `sysTffs.c` in:

```
installDir/vxworks-6.x/target/config/comps /src
```

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 `xxxRegister()` 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.

```plaintext
NOTE: More detailed information on the functionality of each routine is provided in Understanding Socket Driver Functionality, p.158. 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 `f1DelayMsec()` 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 `f1DelayMsec()` 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.158.

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.158 and Socket Windowing and Address Mapping, p.160.

⚠️ 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).

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 = f1SocketOf (noOfDrives);
```

Here, noOfDrives is the running count of drives attached to the system. The routine f1SocketOf() 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 \( V_{pp} \), which is the programming voltage. \( V_{pp} \) is usually 12 Volts to the flash chip. Because not all flash chips require this voltage, the member is included only if \( \text{SOCKET}_12\_\text{VOLTS} \) is defined.

  \( V_{pp} \) must be known to be good on exit and is assumed to be **ON** constantly in RFA environments.

  **NOTE:** The macro \( \text{SOCKET}_12\_\text{VOLTS} \) is only alterable by users that have source code for the TrueFFS core.

- **rfaVppOff()**

  TrueFFS calls this routine to turn off a programming voltage (\( V_{pp} \), usually 12 Volts) to the flash chip. Because not all flash chips require this voltage, the member is included only if \( \text{SOCKET}_12\_\text{VOLTS} \) is defined. This routine is not optional, and must always be implemented. \( V_{pp} \) 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 \( \text{cardDetected} \) to **FALSE** if it is missing.

- **rfaSetWindow()**

  TrueFFS uses \( \text{window.base} \) to store the base address of the memory window on the flash memory, and \( \text{window.size} \) 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*.

**Socket Windowing and Address Mapping**

The `FLSocket` structure (defined in `installDir/vxworks-6.x/target/h/tffs/flsocket.h`) contains an internal window state structure. If you are porting the socket driver, the following background information about this window structure may be useful when implementing the `xxxSetWindow()` and `xxxSetMappingContext()` routines.

The concept of windowing derives from the PCMCIA world, which formulated the idea of a host bus adapter. The host could allow one of the following situations to exist:

- The PCMCIA bus could be entirely visible in the host’s address range.
- Only a segment of the PCMCIA address range could be visible in the host’s address space.
- Only a segment of the host’s address space could be visible to the PCMCIA.

To support these concepts, PCMCIA specified the use of a “window base register” that may be altered to adjust the view from the window. In typical RFA scenarios, where the device logic is NOR, the window size is that of the amount of flash on the board. In the PCMCIA situation, the window size is implementation-specific. The book *PCMCIA Systems Architecture* by Don Anderson provides an explanation of this concept, with illustrations.

**Flash Translation Layer**

This section provides a detailed discussion of the *flash translation layer* (FTL).
Terminology

Due to the complex nature of the FTL layer, you may find the following definitions useful as a reference for the remainder of this section.

**virtual sector number**

The virtual sector number is what the upper software layers see as a sector number. The virtual sector number is used to reference sectors the upper software layers want to access. The virtual sector number can be decoded into page number and sector in page. These decoded numbers are used to find the logical sector number.

**virtual sector address**

The virtual sector address is the virtual sector number shifted left by 9. This gives a byte offset into the flash array corresponding to a 512 byte sector size. A block allocation map (BAM) entry can contain the virtual sector address for logical sectors that contain data.

**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.

**logical sector address**

The logical sector address is the logical sector number shifted left by 9 (512 byte sectors). This address is used for the virtual block map (VBM) entries.
logical sector

If the logical sector term is used, it is referring to the physical sector.

physical sector address

The physical sector address is the memory address in the flash volume where the data that is being referenced by a virtual sector number resides.

physical sector

The physical sector is the data that a virtual sector number references.

garbage sector

This is a physical sector that has been marked as garbage in the BAM. This garbage sector becomes a free sector when a unit transfer happens on the erase unit that contains this garbage sector.

sector in unit

Sector in unit is the number of sectors into the physical erase unit where the physical sector resides.

logical unit number

The logical unit number refers to the number assigned to a physical erase unit when it becomes part of a flash volume. This logical unit number is located in the erase unit header. Transfer units do not have a logical unit number assigned to them. The logical unit number is used to index into the vol.logicalUnits[ ] array. The vol.logicalUnits[ ] array contains pointers into the vol.physicalUnits[ ] array.

logical erase unit

Not all erase units are logical erase units. Only erase units that have been assigned a logical unit number are logical erase units.

physical unit number

The physical unit number can be used to traverse the vol.physicalUnits[ ] array. Generally, the physical unit number is generated when using a logical unit number along with the vol.logicalUnits[ ] array and then using pointer arithmetic off of the pointer into the vol.physicalUnits[ ] array.

physical erase unit

This is the same as an erase unit.

erase unit

An erase unit is the smallest area on the flash device that can be erased. Each erase unit in the flash volume is divided into several physical sectors and includes a header called an erase unit header.

erase unit header

The erase unit header contains information about the FLT volume. It also contains some current information about the erase unit, such as wear-leveling and logical unit number.
virtual block map

A virtual block map (VBM) table is a map of virtual sector numbers to logical sector addresses. Each VBM page takes up one physical sector. The logical sector number of the physical sector that contains a VBM page is stored in the vol.pageTable[ ] array at the time the flash volume is mounted. A virtual sector number is decoded into a page number to be used with the vol.pageTable[ ] array and the sector in page. Each VBM entry contains one of the following: a logical sector address or a designated free sector or deleted sector.

Figure 7-7  Virtual Block Map (VBM)

<table>
<thead>
<tr>
<th>VBM entry</th>
<th>logical sector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 5 2 9 8 7 6 5 4 3 2 1</td>
<td>n/a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>logical unit number</th>
<th>sector in unit</th>
<th>n/a</th>
</tr>
</thead>
</table>

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)

<table>
<thead>
<tr>
<th>BAM entry</th>
<th>depends on the type</th>
<th>block allocation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 5 2 9 8 7 6 5 4 3 2 1</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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.170.) 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.

```c
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.

```c
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 fllite.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 */
    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. */
    LogicalSectorNo * transferUnit;  /* The active transfer unit */
    LogicalSectorNo pageTable;      /* page translation table */
    LogicalSectorNo replacementPageAddress;  /* directly addressable sectors */
    VirtualSectorNo replacementPageSize;
    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 */
    #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 \((ln(\text{vol.flash.erasableBlockSize}) / ln(2))\).

```
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 \((ln(\text{vol.flash.erasableBlockSize}) / ln(2))\)

```
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 tffsFormatParms.

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);
```

noOffPages

noOffPages is the number of VBM pages needed to map all of the virtual
sectors that the upper software layers can access.

```
noOffPages = ((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. Because 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**.

```c
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**.

```c
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.

```c
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.

```c
bamOffset = sizeof(UnitHeader);
```

or this if **formatParams->embeddedCISlength** is not 0

```c
bamOffset = sizeof(UnitHeader) - (sizeof uh->embeddedCIS) +
(formParams->embeddedCISLength + 3) / 4 * 4;
```

**sectorsPerUnit**

**sectorsPerUnit** is the number of physical sectors in an erase unit.

```c
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.

```c
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[ ]** 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[ ]** maps to **&physicalUnits[ ]**. The physical unit number can be
determined using pointer arithmetic off of **&physicalUnits[ ]**.
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.
7.3 Flash File System Support with TrueFFS

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 `ftllite.c`. Note that the erase unit header is only a part of the erase units that are part of the file system. The erase unit header is not part of the erase units in the user reserved area.

```
typedef struct
{
    char formatPattern[15];
    unsigned char numOfTransferUnits; /* no. of transfer units */
    LBulong wearLevelingInfo;
    LBushort logicalUnitNo;
    unsigned char log2SectorSize;
    unsigned char log2UnitSize;
    LBushort firstPhysicalEUN; /* units reserved for boot image */
    LBushort noOfUnits; /* no. of formatted units */
    LBulong virtualMediumSize; /* virtual size of volume */
    LBulong directAddressingMemory; /* directly addressable memory */
    LBushort noOfPages; /* no. of virtual pages */
    unsigned char flags;
    unsigned char eccCode;
    LBulong serialNumber;
    LBulong altEUNoffset;
    LBulong BAMoffset;
    char reserved[12]; /* Actual length may be larger. */
    char embeddedCIS[4]; /* 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.165.

**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()`.

**NOTE:** All values in the unit header are little-endian, thus care is required when reading or writing to this table.
static char FORMAT_PATTERN[15] = {0x13, 3, 'C', 'I', 'S',
                                  0x46, 57, 0, 'F', 'T', 'L', '1', '0', '0', 0};

noOfTransferUnits
The noOfTransferUnits is set by the user when the flash volume is formatted.
For more information, refer to noOfSpareUnits in tffsFormatParams.
noOfTransferUnits = formatParams->noOfSpareUnits;

wearLevelingInfo
wearLevelingInfo is used to keep track wear-leveling for the flash volume.
This parameter is specific to each erase unit. This allows the FTL to keep track
of the order in which erase units are erased. When an erase unit is erased,
wearLevelingInfo increments the value of vol.currWearLevelingInfo.
vol.currWearLevelingInfo keeps track of the number of times any of the erase
units have been erased on a flash volume.

logicalUnitNo
logicalUnitNo is the logical unit number of an erase unit. This number is
unique for each logical erase unit. logicalUnitNo is used as an index into
vol.logicalUnits[ ] array. When mounting a flash volume, the physical unit
associated with &vol.physicalUnits[ ] is assigned to
vol.logicalUnits[logicalUnitNo]. If the erase unit is not mapped into the
vol.logicalUnits[ ] array, logicalUnitNo has a value of
UNASSIGNED_UNIT_NO (0xffff) or MARKED_FOR_ERASE (0x7fff). If the
value is UNASSIGNED_UNIT_NO the erase unit is a transfer unit. If the value
is MARKED_FOR_ERASE then the erase unit is in the process of a unit transfer.
For a very short period of time during a unit transfer there can be 2 erase units
with the same logicalUnitNo.

log2SectorSize
log2SectorSize is the number of bits needed to store the physical sector size.
Because the sector size is 512 bytes, the calculation is (\ln(512) / \ln(2)) which is
9.

#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.
8.1 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.

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*.
9.1 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.183.

9.3.1 {sioChanGet}()

The func{sioChanGet}() routine retrieves the SIO_CHAN structure associated with the specified serial port.

```c
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.
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:

```c
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.

```c
#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.183.

For additional information on serial driver initialization, see *9.11 Serial Drivers, Initialization, and Interrupts*, p.185.
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

The structures are defined 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
    );
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_AVAL_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_AVAL_MODES_GET is used, the values of SIO_MODE_POLL and SIO_MODE_INT are logically or-d together as follows:

  ```c
  *(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). 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.

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*. 
10.1 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.192 and in 10.7 Initialization, p.191.

### 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.

```c
#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.192.

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( ) deregisters the routine registered by erfHandlerRegister().

For more information on these routines, see ERF Registration, p.192.

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.195.

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.194).
- (*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.193.

bio_done( )

The bio_done( ) routine is used to mark the bio structure as processed. For more information, see 10.8.2 Processing, p.194.

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 (\texttt{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.

For more information on the XBD facility, see the *VxWorks Kernel Programmer’s Guide: I/O System*.

### 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 \texttt{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:

```c
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()`:

```c
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:

```c
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)()` 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:

```c
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:

```c
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*. 
11.1 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 runtime 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:

```c
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. For the type definition for `vxbTimerFunctionality`, refer to the following header file:

```
installDir/vxworks-6.x/target/h/vxbTimerLib.h
```

This file also includes 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.205), the driver sets this field to `TRUE`. When a timer is released (using the `(*timerRelease)()` function pointer, see 11.9.2 `(*timerRelease)()`, p.205), 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 field 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.205).
- `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.208) 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.205:

- 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 (\texttt{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 (\texttt{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.

The two principal elements to this data structure are the \texttt{VXB\_DEVICE\_ID} instance data (by convention referred to as \texttt{pInst}), and the driver-specific data structure, which in this figure is labeled as \texttt{struct myTimerData}. Note that each of these data structures contains a pointer to the other; the \texttt{pInst->pDrvCtrl} field contains a pointer to the driver-specific data structure, and the driver-specific data structure contains a pointer back to the \texttt{pInst}.

When a timer driver initializes itself, it typically allocates its \texttt{struct myTimer} using \texttt{hwMemAlloc()} , and then initializes the various data structures contained within it. This includes initializing the pointer(s) to \texttt{pInst}. Because the \texttt{VXB\_DEVICE\_ID} \texttt{pInst} pointer is not provided to the service routines, the stored pointer(s) to \texttt{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:

```c
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:

```c
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.206). Timestamp drivers are required to support this routine. This routine is not used for periodic interrupt drivers.

The prototype for this routine is:

```c
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:

```c
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.

The prototype for this routine is:

```c
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.

The prototype for this routine is:

```c
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.

NOTE: If the timer unit is always used as a free running counter—such as a timestamp timer or a delay timer—and the timer unit has the following characteristics:

- It starts at power on with the maximum count value supported by the hardware.
- The counter value is automatically rolled over when it reaches the maximum count.

This routine can return OK with no other operations. (For an example, see the vxbIntelTimestamp driver.)

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:

```c
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:

```c
(*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:

```c
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:

```c
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:

```c
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.
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 = SYSCLK_TIMER_NAME;`
- `UINT32 sysClkDevUnitNo = SYSCLK_TIMER_UNIT;`
- `UINT32 sysClkTimerNo = SYSCLK_TIMER_NUM;`

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:

```c
char * pAuxClkName = AUXCLK_TIMER_NAME;
UINT32 auxClkDevUnitNo = AUXCLK_TIMER_UNIT;
UINT32 auxClkTimerNo = AUKCLK_TIMER_NUM;
```

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*.
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.218.

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, `{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 `{vxbDrvUnlink}()` routine performs the disconnect by calling the remove routine provided by each class driver for each device connected to the hub.

```c
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`.

```c
#include <usb/usbOsal.h>
```

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`.

```c
#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.

```c
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.

```c
const struct hcfDevice hcfDeviceList[] = {
    ...
    { "vxbPlbUsbXXXX", 0, VXB_BUSID_PLB, 0, vxbPlbUsbXXXDevNum0, vxbPlbUsbXXXDevResources0},
    { "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.

```c
const struct hcfResource vxbPlbUsbXXXDevResources0 [] = {
    { "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(vxbPlbUsbXXXDevResources0)
```

**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.

```c
/* **************************************************************************
 * 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.

### 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:

```bash
-> usbInit
value = 0 = 0x0
```

Initialize the EHCI host controller driver:

```bash
-> usbEhcdInit
value = 0 = 0x0
```
Register the EHCI driver with VxBus:

```c
-> vxbUsbEhciRegister
value = 0 = 0x0
```

List the USB host controllers and devices:

```c
-> 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 0x004555604 on bus Local_Bus, funcs @ 0x0045559c
vxbUsbEhciHub at 0x0045555c4 on bus USB-EHCI_Bus, funcs @ 0x004555a8
vxbUsbHubClass at 0x0048c198 on bus USB-HUB_Bus, funcs @ 0x004556ec
.
.
Registered Busses and Devices Present:
Local_Bus @ 0x00471b70 with bridge @ 0x0044f718
Device Instances:
s16550 unit 0 on Local_Bus @ 0x00472b30 with busInfo 0x00000000
s16550 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 0x004777b0
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

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 `func{driverControl}()`
with the appropriate functionality name and driver name. For an example of how to use this mechanism, see the sample driver in:

\texttt{installDir/vxworks-6.x/target/3rdparty/windriver/wrsample}

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 \textit{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.
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.

BAR

Base address register.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*busCfgRead)( )</td>
<td>10</td>
</tr>
<tr>
<td>overriding 10</td>
<td></td>
</tr>
<tr>
<td>(*busCfgWrite)( )</td>
<td>10</td>
</tr>
<tr>
<td>overriding 10</td>
<td></td>
</tr>
<tr>
<td>(*dmaCancel)( )</td>
<td>30</td>
</tr>
<tr>
<td>(*dmaPause)( )</td>
<td>30</td>
</tr>
<tr>
<td>(*dmaRead)( )</td>
<td>29</td>
</tr>
<tr>
<td>(*dmaReadAndWait)( )</td>
<td>29</td>
</tr>
<tr>
<td>(*dmaStatus)( )</td>
<td>29</td>
</tr>
<tr>
<td>(*dmaWrite)( )</td>
<td>29</td>
</tr>
<tr>
<td>(*dmaWriteAndWait)( )</td>
<td>29</td>
</tr>
<tr>
<td>(*timerAllocate)( )</td>
<td>205</td>
</tr>
<tr>
<td>(*timerCountGet)( )</td>
<td>206</td>
</tr>
<tr>
<td>(*timerCountGet64)( )</td>
<td>210</td>
</tr>
<tr>
<td>(*timerDisable)( )</td>
<td>207</td>
</tr>
<tr>
<td>(*timerEnable)( )</td>
<td>208</td>
</tr>
<tr>
<td>(*timerEnable64)( )</td>
<td>209</td>
</tr>
<tr>
<td>(*timerISRSet)( )</td>
<td>208</td>
</tr>
<tr>
<td>(*timerRelease)( )</td>
<td>205</td>
</tr>
<tr>
<td>(*timerRolloverGet)( )</td>
<td>206</td>
</tr>
<tr>
<td>(*timerRolloverGet64)( )</td>
<td>209</td>
</tr>
<tr>
<td>(*vxbDevControl)( )</td>
<td>10</td>
</tr>
<tr>
<td>overriding 10</td>
<td></td>
</tr>
<tr>
<td>(*xf_dump)( )</td>
<td>191, 194</td>
</tr>
<tr>
<td>(*xf_ioctl)( )</td>
<td>191, 193, 195</td>
</tr>
<tr>
<td>(*xf_strategy)( )</td>
<td>191, 194</td>
</tr>
<tr>
<td>{busCtlrAccessOverride}( )</td>
<td>9</td>
</tr>
<tr>
<td>{busCtlrBaseAddrCvt}( )</td>
<td>11</td>
</tr>
<tr>
<td>{busCtlrCfgInfo}( )</td>
<td>11</td>
</tr>
<tr>
<td>{busCtlrCfgRead}( )</td>
<td>6</td>
</tr>
<tr>
<td>{busCtlrCfgWrite}( )</td>
<td>7</td>
</tr>
<tr>
<td>{busCtlrDevCfgRead}( )</td>
<td>5</td>
</tr>
<tr>
<td>{busCtlrDevCfgWrite}( )</td>
<td>7</td>
</tr>
<tr>
<td>{busCtlrDevCtrl}( )</td>
<td>8</td>
</tr>
<tr>
<td>{busDevShow}( )</td>
<td>46</td>
</tr>
<tr>
<td>{cpmCommand}( )</td>
<td>176</td>
</tr>
<tr>
<td>{driverControl}( )</td>
<td>176, 226</td>
</tr>
<tr>
<td>{isrRerouteNotify}( )</td>
<td>57</td>
</tr>
<tr>
<td>{m85xxLawBarAlloc}( )</td>
<td>176</td>
</tr>
<tr>
<td>{miiBusRead}( )</td>
<td>130</td>
</tr>
<tr>
<td>{miiLinkUpdate}( )</td>
<td>131, 135</td>
</tr>
<tr>
<td>{miiMediaUpdate}( )</td>
<td>74, 77, 89</td>
</tr>
<tr>
<td>{miiModeGet}( )</td>
<td>88, 132</td>
</tr>
<tr>
<td>{miiModeSet}( )</td>
<td>89, 131</td>
</tr>
<tr>
<td>{miiRead}( )</td>
<td>74, 78, 88, 89, 131, 134, 135</td>
</tr>
<tr>
<td>{miiWrite}( )</td>
<td>74, 78, 88, 131, 135</td>
</tr>
<tr>
<td>{mux2DevConnect}( )</td>
<td>74, 78, 91, 122, 125</td>
</tr>
<tr>
<td>{muxConnect}( )</td>
<td>135</td>
</tr>
<tr>
<td>{muxDevConnect}( )</td>
<td>74, 89, 91, 122, 125</td>
</tr>
<tr>
<td>{muxDevConnect2}( )</td>
<td>75</td>
</tr>
<tr>
<td>{nonVolGet}( )</td>
<td>138</td>
</tr>
<tr>
<td>{nonVolSet}( )</td>
<td>138</td>
</tr>
<tr>
<td>{sioChanConnect}( )</td>
<td>181</td>
</tr>
<tr>
<td>{sioChanGet}( )</td>
<td>180</td>
</tr>
<tr>
<td>{vxbDevRegMap}( )</td>
<td>12</td>
</tr>
<tr>
<td>{vxbDmaResDedicatedGet}( )</td>
<td>27</td>
</tr>
<tr>
<td>{vxbDmaResourceGet}( )</td>
<td>26</td>
</tr>
<tr>
<td>{vxbDmaResourceRelease}( )</td>
<td>27</td>
</tr>
<tr>
<td>{vxbDrvUnlink}( )</td>
<td>74, 76, 121, 125, 219</td>
</tr>
<tr>
<td>{vxbIntCtrlConnect}( )</td>
<td>36</td>
</tr>
<tr>
<td>{vxbIntCtrlCpuReroute}( )</td>
<td>38, 56</td>
</tr>
<tr>
<td>{vxbIntCtrlDisable}( )</td>
<td>37</td>
</tr>
<tr>
<td>{vxbIntCtrlDisconnect}( )</td>
<td>36</td>
</tr>
<tr>
<td>{vxbIntCtrlEnable}( )</td>
<td>37</td>
</tr>
<tr>
<td>{vxbIntCtrlIntReroute}( )</td>
<td>38</td>
</tr>
<tr>
<td>{vxbIntDynaVecConnect}( )</td>
<td>37, 53</td>
</tr>
<tr>
<td>{vxbIntDynaVecProgram}( )</td>
<td>16, 54</td>
</tr>
<tr>
<td>{vxbTimerFuncGet}( )</td>
<td>200</td>
</tr>
<tr>
<td>{vxIpiControlGet}( )</td>
<td>39, 58</td>
</tr>
</tbody>
</table>

### Numerics

- 00tfs.cdf 152
- 802.1q VLAN lag extraction 107

### A

- access routine 229
- accessing MII management registers 78
address translation
  DMA  87
  for USB drivers  220
  in bus controller drivers  11
advertise
  definition  229
advertising XBD methods  193
allocating
  device structures  65
  DMA channels  26
  system resources  175
  timers  201, 205
  tuples from the pool  87
AMD/Fujitsu flash devices
  CFI  145
  non-CFI  146
amdmtd.c  146
analog-to-digital converters  225
announcing
  a downstream bus  20
  devices to VxBus  23, 65
ATA commands  196
ATA_RESOURCE  190
ATA_TYPE  190
ataResources  190
ataTypes  190
autoIntRouteSet  18
auxiliary clock  213

B

BAM  142, 163
BAR  229
binding a network device to the stack  74, 75
bio  191, 194
bio.h  194
bio_done()  191, 194
block allocation map, see BAM
boot block  145
bridgePostConfigFuncSet  18
bridgePreConfigFuncSet  18
BSP configuration
  bus controller drivers  17
  CPU routing table  42
  DMA drivers  28
  dynamic vector table  42
  dynamic vectors  53
  interrupt controller drivers  40
  interrupt input table  40
  interrupt priority  43
  MAC drivers  79
  multifunction drivers  64
  NVRAM drivers  139
  other class drivers  227
  PHY drivers  133
  resource drivers  176
  resources for a PCI bus  17
  serial drivers  182
  storage drivers  190
  timer drivers  203
  USB drivers  220
BSP resources
  ataResources  190
  ataTypes  190
  clkFreq  203
  clkRateMax  203
  clkRateMin  203
  cpuClkRate  203
  for PCI autoconfiguration  18
  inputTableSize  41
  miiIfName  80
  miiIfUnit  80
  numSegments  139
  phyAddr  80
  priority  43
  priorityTableSize  43
  segments  139
buffer management  83
  IPNET-native drivers  83
  M_BLK-style drivers  86
BUFFER_WRITE_BROKEN  144
bus
  definition  229
  bus controller drivers  3
    address translation  11
    BSP configuration  17
    communication  4
    debugging  24
    deferring driver initialization  24
    driver methods  5
    generating configuration transactions  11
    header files  16
    initialization  21
    initialization example  22
    location in VxWorks  4
    overriding service routines  9
    overview  3
    utility routines  19
bus controllers
  connecting to devices  20
  definition  229
bus discovery  229
bus match  229
bus type
  definition  229
  macros  20
  registering with VxBus  23

C

cacheSize  18
callbackInstall()  184
calling the socket registration routines  155
cancelling an operation on a DMA channel  30
cap_available 100
cap_enabled 99, 107
CDFS
  00tffs.cdf 152
CFI 143
  AMD/Fujitsu command set 143
  AMD/Fujitsu flash devices 145
  Intel/Sharp command set 143
CFI/SCS flash support 144
CFI_DEBUG 144
cfiamd.c 143
cfiscs.c 143, 144
checksum offload
  M_BLK-style drivers 107
  per-packet interface
    for IPNET-native drivers 103, 105
child 230
classes, see driver classes
clkFreq 203
clkRateMax 203
clkRateMin 203
CLOCAL 184
cluster 96, 230
common flash interface, see CFI
communication, bus controller 4
components
  adding MTD components 152
  INCLUDE_GENERICPHY 130, 133
  INCLUDE_GENERICTBIPHY 130, 133
  INCLUDE_MII_BUS 127, 133
  INCLUDE_NON_VOLATILE_RAM 139
  INCLUDE_PARAM_SYS 133
config.h
  MTD identification 154
configuring
  BSP resources for PCI bus 17
  interrupt controllers 45
connecting
  an ISR to a timer 208
  bus controllers to devices 20
  ISRs 36
  networking code 90
copying data to and from NVRAM 138
CPU routing table 42
cpuClkRate 203
CREAD 184
creating
  a new bus 66
  an MII bus instance 88
  an XBD 192
  buffer pools 87
crossbar routing table 43
CSIZE 184
CSUM_DATA_VALID 101, 108
CSUM_DELAY_DATA 101
CSUM_DELAY_Data6 101
CSUM_DELAY_IP 101
csum_flags_rx 100
csum_flags_tx 100, 103, 104, 109
CSUM_FRAGMENT 101
CSUM_IP 100, 109
CSUM_IP_CHECKED 101, 108
CSUM_IP_FRAGS 101
CSUM_IP_HDRLEN 110
CSUM_IP_VALID 101, 108
CSUM_IPHDR_LEN 102
CSUM_IPHDR_OFFSET 102, 110
CSUM_PSEUDO_HDR 101, 108
CSUM_PTAGGED 101
CSUM_RESULTS 102
CSUM_TCP 100, 103, 109
CSUM_TCP_SEG 101
CSUM_TCPv6 100, 103, 109
CSUM_TCPv6_SEG 101
CSUM_UDP 100, 103, 109
CSUM_UDPv6 100, 109
CSUM_VLAN 100, 104
CSUM_XPORT_CSUM_OFF 102, 110
custom drivers 225
data structures, see structures
data_freefunc_cookie 86
DEBUG_PRINT 144
deferring
  driver registration for MAC drivers 119
  ISRs 91
defining MTDs
  as VxWorks components 152
  in the socket driver file 153
deleting an MII bus 88
descriptor 230
devices
  accessing MII management registers 78
  AMD/Fujitsu flash, CFI 145
  AMD/Fujitsu flash, non-CFI 146
  CFI
creating an XBD 192
definition 230
enumeration 23
flash 137, 143
Intel 28F008 flash 145
Intel 28F016 flash 145
interactions on multifunction chips 66
interleaved registers 66
multiple devices on a single chip 63
NAND 146
network device initialization 89
PHY 87
device probing and discovery 128
registering with VxBus 23
shared resources 67
timers 199
devInstanceInit2() 83, 85
devResourceGet() 140
digital-to-analog converters 225
direct memory access drivers, see DMA drivers
disabling
interrupt inputs 36
ISRs 37, 45	
timer interrupt generation 207
discarding
devices 23
PHY devices 128
disk-on-chip support 146
dispatching
interrupts 47
ISRs 50
DMA
genes 71
support 87
support for IPNET-native drivers 87
DMA channel
cancelling an operation on 30
generating status of 30
pausing a 30
queueing a read operation 29
queueing a write operation 29
DMA drivers 25
BSP configuration 28
debugging 31
driver methods 26
header files 27
initialization 28
overview 25
structures and routines 28
utility routines 28
DMA_IDLE 30
DMA_NOT_USED 30
DMA_PAUSED 30
DMA_RUNNING 30
documentation about 2
downstream 230
driver
definition 230
driver classes 1
bus controller 3
direct memory access 25
interrupt controller 33
multifunction 63
network 69
network interface (MAC) 71
NVRAM 138
other 225
PHY 127
resource 67, 175
serial 179
timer 199
USB 217
driver methods
|busCtlrAccessOverride() 9
|busCtlrBaseAddrCvt() 11
|busCtlrCfgInfo() 11
|busCtlrCfgRead() 6
|busCtlrCfgWrite() 7
|busCtlrDevCfgRead() 5
|busCtlrDevCfgWrite() 7
|busCtlrDevCtrl{} 8
|busDevShow{} 46
|cpmCommand{} 176
|driverControl{} 176, 226
|isrRerouteNotify{} 57
|m855xxLawBarAlloc{} 176
|miiBusRead{} 130
|miiLinkUpdate{} 131, 135
|miiMediaUpdate{} 74, 77, 89
|miiModeGet{} 88, 132
|miiModeSet{} 89, 131
|miiRead{} 74, 78, 88, 89, 131, 134, 135
|miiWrite{} 74, 78, 88, 131, 135
|mux2DevConnect{} 74, 89, 91, 122, 125
|muxConnect{} 135
|muxDevConnect{} 74, 89, 91, 122, 125
|muxDevConnect2{} 75
|nonVolGet{} 138
|nonVolSet{} 138
|sioChanConnect{} 181
|sioChanGet{} 180
|vxbDevRegMap{} 12
|vxbDmaResDedicatedGet{} 27
|vxbDmaResourceGet{} 26
|vxbDmaResourceRelease{} 27
|vxbDrvUnlink{} 74, 76, 121, 125, 219
|vxbIntCtrlConnect{} 36
|vxbIntCtrlCpuReroute{} 38, 56
|vxbIntCtrlDisable{} 37
|vxbIntCtrlDisconnect{} 36
|vxbIntCtrlEnable{} 37
|vxbIntCtrlIntReroute{} 38
|vxbIntDynaVecConnect{} 37, 53
|vxbIntDynaVecProgram{} 16, 54
|vxbTimerFuncGet{} 200
|vxIpiControlGet{} 39, 58
bus controller drivers 5
definition 230
DMA drivers 26
interrupt controller drivers 36
MAC drivers 74
Index

multifunction drivers 64
multiprocessor systems 38
NVRAM drivers 138
other class drivers 226
PHY drivers 130
resource drivers 176
serial drivers 180
storage drivers 190
timer drivers 200
USB drivers 219
driverAccessFunc() 15
DRV_CTRL 22
dynamic vector assignment 37
dynamic vector table 42
dynamic vectors 35
configuring
in the BSP 53
using service driver routines 53
determining values for 54
interrupt assignment 16
programming 54

E

Enhanced Host Controller Interface, see EHCI
enhanced network drivers, see END drivers
enumeration 230
device 23
erasableBlockSize 152
erase units 170
ERF 192
event reporting 195
event types 195
new device notification 194
registering 192
ERF_ASYNC_PROC 194
ERF_SYNC_PROC 194
erfEventRaise() 191, 194, 195
erfHandlerRegister() 191, 193
erfHandlerUnregister() 191, 193
Ethernet 70
event reporting framework, see ERF
extended block device, see XBD
EXTRA_INCLUDE 39

F

ffsEnable 18
files
00ttfs.cdf 152
amdmtd.c 146
cfiamd.c 143
cfiscs.c 143, 144
config.h 154
donCommon.h 98
donLib.c 86
hwconf.c 40, 133
i28F008.c 145
i28F016.c 145
ipnet_eth.h 107
mbuf.h 100
miiBus.c 87
miiBus.o 127
sysTffs.c 144, 145, 146, 152, 153, 154, 155
tfsConfig.c 147, 154
usbPciStub.c 220
vxbAuxClkLib.c 213
vxbIntClkLib.c 46
vxbIntellchStorage.c 196
vxbPci.c 19
vxbSd31xxStorage.c 196
vxbSysClkLib.c 211
vxbTimestampLib.c 214
vxBus.c 65
vxmux_config.h 92
vxmux_pkt.h 92
finding interrupt inputs 45
flash
device layout 170
erase units 170
interleaved chips 170

237
flash file system support, see TrueFFS
flash translation layer, see FTL
FLASH_BASE_ADRS 157
FLASH_SIZE 157
flbase.h 149
flDelayMsec() 156
flDontNeedVpp() 150
FLFlash 147, 152
flflash.h 147
flNeedVpp() 150
flSetWindowSize() 157
FLSocket 149, 156, 160
flSocketOf() 158
flsystem.h 152
flWriteProtected() 150
freeing a DMA channel 27
FTL 142, 160
overview 164
structures 165
terminology 161
function pointers
(*busCfgRead)() 10
(*busCfgWrite)() 10
(*dmaCancel)() 30
(*dmaPause)() 30
(*dmaRead)() 29
(*dmaReadAndWait)() 29
(*dmaStatus)() 30
(*dmaWrite)() 29
(*dmaWriteAndWait)() 29
(*timerAllocate)() 205
(*timerCountGet)() 206
(*timerCountGet64)() 210
(*timerDisable)() 207
(*timerEnable)() 208
(*timerEnable64)() 209
(*timerISRSet)() 208
(*timerRelease)() 205
(*timerRolloverGet)() 206
(*timerRolloverGet64)() 209
(*vxbDevControl)() 10
(*x_dump)() 191, 194
(*x_ioctl)() 191, 193, 195
(*x_strategy)() 191, 194
vxblntDynaCtrlInputInit() 48
vxblntDynaVecProgram() 48

getting
an ISR function pointer 46
arguments for an ISR 46
flags for an interrupt input 46
status of a DMA channel 30

H

handling
corrupt packets with M_BLK-style drivers 108
interrupts 91
network interrupts 73
HCDs 218
HCF_RES_ADDR 17, 41
HCF_RES_INT 17, 18, 41
header files
bio.h 194
bus controller drivers 16
DMA drivers 27
dndMedia.h 131, 132
flbase.h 149
flFlash.h 147
flsystem.h 152
genericTbiPhy.h 130
interrupt controller drivers 39
MAC drivers 79
multifunction drivers 64
NVRAM drivers 139
other class drivers 227
PHY drivers 133
resource drivers 176
serial drivers 181
sioLib.h 180, 183
sioLibCommon.h 184
storage drivers 190, 203
USB drivers 219
usbHst.h 219
usbOsal.h 219
vxbAccess.h 9, 14
vxbDmaDriverLib.h 26
vxbDmaLib.h 28
vxbIntCtrlLib.h 39
vxbIntrCtrl.h 39
vxbNonVol.h 139
vxbTimerLib.h 201
vxBus.h 20, 64
xbd.h 193
heartbeat interrupt 200
hEND drivers, see hierarchical END drivers
HEND_RX_QUEUE_PARAM 82
hierarchical END drivers 136
host controller drivers, see HCDs
HUPCL 184

G

generating
bus controller configuration transactions 11
dynamic vectors 47
genericPhy 127, 128, 130, 132
genericTbiPhy 130
genericTbiPhy.h 130
Index

hwconf.c 45, 133
   interrupt controller resources 40
   NVRAM 139
   USB drivers 220
hwMemAlloc() 22, 204

I
I28F008.c 145
i28f016.c 145
identifying interrupts 34
IFCAP_CAP0 99
IFCAP_CAP1 99
IFCAP_CAP2 99
IFCAP_CAP3 99
IFCAP_IPCOMP 99
IFCAP_IPSEC 99
IFCAP_JUMBO_MTU 99
IFCAP_NETCONS 99
IFCAP_RXCSUM 99, 105, 107
IFCAP_RXSUM 102
IFCAP_TCPSEG 99
IFCAP_TXCSUM 99
IFCAP_VLAN_HWTAGGING 99, 104, 107
IFCAP_VLAN_MTU 99
ifconfig() 119, 121
IFM_ACTIVE 132
IFM_AUTO 89, 134
IFM_AVALID 132
implementing
timer driver service routines 205
   VxWorks auxiliary clock 213
   VxWorks system clock 211
   VxWorks timestamp drivers 214
in_cksum_skip() 110
INCLUDE_EHCI 222
INCLUDE_END2 126
INCLUDE_END2_LINKBUFFPOOL 126
INCLUDE_GENERICPHY 130, 133
INCLUDE_GENERICTBIHY 130, 133
INCLUDE_MI_BUS 127, 133
INCLUDE_MTD_AMD 146
INCLUDE_MTD_CFISC 144
INCLUDE_MTD_I28F008 146
INCLUDE_MTD_I28F016 145
INCLUDE_NON_VOLATILE_RAM 139
INCLUDE_OHCI 222
INCLUDE_PARAM_SYS 133
INCLUDE_UHCI 222
INCLUDE_USB_INIT 222
INCLUDE_WDB_COMM_END 126
includeFuncSet 18
initializing
   a network 119
   bus controller drivers 21
   DMA drivers 28
   FLFlash structure members 147
interrupt controller drivers 48
MAC drivers 89
multifunction drivers 66
network devices 89
NVRAM drivers 140
other class drivers 227
PHY drivers 135
resource drivers 177
serial drivers 182, 186
storage drivers 191
timer drivers 203
USB drivers 221
inputTableSize 41
instance
definition 230
intAssignFuncSet 18
intCpuUnlock() 51
intCtrlChainISR() 46, 47
intCtrlCpu 42
intCtrlHwConfGet() 44, 45
intCtrlHwConfShow() 44, 46
intCtrlISRAdd() 44, 45
intCtrlISRDisable() 44, 45
intCtrlISREnable() 44, 45
intCtrlISRRemove() 44, 45
intCtrlPinFind() 44, 45, 54
intCtrlStrayISR() 46, 47
intCtrlTableArgGet() 44, 46
intCtrlTableCreate() 44, 46
intCtrlTableFlagsGet() 44, 46
intCtrlTableFlagsSet() 44, 46
intCtrlTableISRGet() 44, 46, 47
intCtrlTableUserSet() 44, 47
integrating a timer driver with VxWorks 211
Intel 28F008 flash devices 145
Intel 28F016 flash devices 145
Intel ICH storage driver 192
interaction
   PHY and MII bus 87
   serial ports and WDB connection 186
interleaved registers 66
INTERLEAVED_MODEQUIRES_32BIT_WRITES 144
interprocessor interrupts, see IPIs
interrupt controller drivers 33
   BSP configuration 40
   CPU routing table 42
crossbar routing table 43
debugging 61
dispatch routines 47
driver methods 36
dynamic vector assignment 35, 37
dynamic vector table 42
header files 39
initialization 48
interrupt input table 40
interrupt priority 43, 49
managing dynamic vectors 52
multiprocessing 36, 38, 56
<table>
<thead>
<tr>
<th>Interrupt Controllers</th>
<th>OpenPIC 34 overview 34 programming dynamic vectors 54 releasing third-party drivers 39 responsibilities 34 typologies 49 utility routines 44 vxbEpicIntCtlr.c 34 vxbPpcIntCtlr.c 34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Inputs</td>
<td>responsibilities 34 typologies 49 utility routines 44 vxbEpicIntCtlr.c 34 vxbPpcIntCtlr.c 34</td>
</tr>
<tr>
<td>Interrupt-Driven Mode</td>
<td>serial drivers 183</td>
</tr>
<tr>
<td>Interrupts</td>
<td>connecting ISRs 45 disabling interrupt inputs 36 ISRs 45 dispatching 47 dynamic vector assignment 16, 35, 37 dynamic vector management 47 dynamic vector table 42 enabling 47 interrupt inputs 37 ISRs 45 finding inputs 45 getting an ISR function pointer for flags 46 handlers 91 identifying 34 input table 40 interrupt controller drivers 33 managing dynamic vectors 52 network 73 PHY 127 priority 43, 49 programming dynamic vectors 54 removing an ISR 45 rerouting 38 retrieving ISR arguments 46 routing in an SMP system 56 routing to a specific CPU 57 serial drivers 186 setting flags in isrHandle 46 transmit-packet-complete 73 validating 120</td>
</tr>
<tr>
<td>IntrCtlrCpu</td>
<td>57</td>
</tr>
<tr>
<td>IntrCtlrInputs</td>
<td>40</td>
</tr>
<tr>
<td>IntrCtlrPriority</td>
<td>43</td>
</tr>
<tr>
<td>io16Addr</td>
<td>17, 18</td>
</tr>
<tr>
<td>io16Size</td>
<td>17, 18</td>
</tr>
<tr>
<td>io32Addr</td>
<td>17, 18</td>
</tr>
<tr>
<td>io32Size</td>
<td>17, 18</td>
</tr>
<tr>
<td>ioctl()</td>
<td>184</td>
</tr>
<tr>
<td>iodesc</td>
<td>15</td>
</tr>
<tr>
<td>-IP_ERRNO_EWOULDBLOCK</td>
<td>95</td>
</tr>
<tr>
<td>ipcom_drv_eth_init()</td>
<td>119</td>
</tr>
<tr>
<td>IPCOM_FLAG_FC_EXACT</td>
<td>84</td>
</tr>
<tr>
<td>Ipcom_pkt</td>
<td>84, 87, 92</td>
</tr>
<tr>
<td>Ipcom_pkt structure</td>
<td>72</td>
</tr>
<tr>
<td>IPCOM_PKT_FLAG_HW_CHECKSUM</td>
<td>103, 106</td>
</tr>
<tr>
<td>IPCOM_PKT_FLAG_TL_CHECKSUM</td>
<td>106</td>
</tr>
<tr>
<td>IPCOM_PKT_FLAG_VLAN_TAG</td>
<td>104</td>
</tr>
<tr>
<td>IPs</td>
<td>57</td>
</tr>
<tr>
<td>managing 39 IPNET packet pool 83 ipnet_eth.h 107 IPNET_ETH_PKT_SET_VLAN_TAG 107 IPNET_PKT_ALIGNMENT 93 IPNET-native drivers see also MAC drivers 802.1q VLAN tag extraction 107 allocating packet buffers 92 binding a device to the stack 75 buffer management 83 DMA support 87 overview 71 per-packet receive checksum offload interface 105 per-packet transmit checksum offload interface 103 polled mode 126 support for scatter-gather 95 working with Ipcom_pkts 92 isrDeferIsrReroute() 57 isrDeferLib 57 isrHandle 45, 55 printing contents of 46 setting flags in 46 isrRerouteNotify() 57 ISRs 91 calling 47 connecting 36 to an interrupt 45 to timer hardware 208 deferring 91 disabling 37, 45 dispatching 50 enabling 45 function pointers 46 removing 45 retrieving arguments to 46</td>
<td></td>
</tr>
</tbody>
</table>

J

job queueing 82 jobQueueCreate() 83 jobQueueInit() 83 jobQueueLib 80, 82, 91 jobQueuePost() 82, 92, 110 jobQueueStdPost() 82 jumbo frames, support for 86
Index

L

ld( ) 125
libraries
  endLib 87
  isrDeferLib 57
  jobQueueLib 80, 82, 91
  loadLib 125
  muxLib 80, 81
  netBufLib 80, 83, 86, 96
  utility library for PCI configuration 17
  vxbAuxClkLib 213
  vxbDmaBufLib 80, 87
  vxbDmaEnd2BufLib 87
  vxbDmaLib 26, 28
  vxbIntCtrlrLib 35, 44, 47, 55
  vxbSysClkLib 211
  vxbTimestampLib 214
  vxpLib 58
  loadLib 125
  logMsg( ) 120
lower edge methods 131
lower edge utility routines 133

M

M_BLK-style drivers
  see also MAC drivers
  buffer management 86, 96
  checksum offload 107
  checksum offloading send routine 109
  CSUM flags 109
  driver receive routine 107
  handling corrupt packets 108
  setting up a memory pool 96
  support for scatter-gather 97
M_PKT_HDR 101
MAC drivers 69
  attaching
    to the IPv4 stack 119
to the MUX 119
  binding a device to the stack 74
  BSP configuration 79
  command and control module 73
  connecting networking code 90
  debugging 118
    with show routines 118
deferring driver registration 119
  driver methods 74
  functional modules 72
  header files 79
  initialization 89
  interrupt handlers 91
  interrupts 73
  loading and unloading 121
  lower edge methods 131
  lower edge utility routines 133
multicast filter test 127
overview 71
pairing with a PHY instance 120
PHY and MII bus interactions 87
ping-of-death test 121
polled mode testing 125
protocol impact on 98
receive error path testing 126
receive handling method 110
receive stall handling 118
reception module 72
relationship to MII bus 129
starting and stopping 121
stress testing 120
terminating an instance 76
testing with Netperf 120
transmission module 73
upper edge methods 131
upper edge utility routines 133
utility routines 80
validating interrupts 120
WTX test 126
macros
  BUFFER_WRITE_BROKEN 144
  CFI_DEBUG 144
  DEBUG_PRINT 144
  INCLUDE_EHCI 222
  INCLUDE_MTD_AMD 146
  INCLUDE_MTD_CFI5CS 144
  INCLUDE_MTD_I28F008 146
  INCLUDE_MTD_I28F016 145
  INCLUDE_OHCI 222
  INCLUDE_UHCI 222
  INTERLEAVED_MODE_REQUIRES_32BIT_WRITES 144
  SAVE_NVRAM_REGION 144
  VXB_BUSID_MII 20
  VXB_BUSID_PCI 20
  VXB_BUSID_PNO 20
  VXB_BUSID_RAPIDIO 20
  VXB_BUSID_VIRTUAL 20
  VXB_HANDLE( ) 15
  VXB_HANDLE_WIDTH( ) 15
  VXB_INTCTLR_ISR_CALL( ) 44, 47, 51
  VXB_INTCTLR_PINENTRY_ALLOCATED( ) 44, 47, 56
  VXB_INTCTLR_PINENTRY_ENABLED( ) 44, 47
  VXB_INTCTLR_PINENTRY_LOWLEVEL_SIZE 55
  VXB_INTCTLR_PINENTRY_TOPLEVEL_SIZE 55
  VXB_LEGACY_ACCESS 6, 7, 9
managing
dynamic vectors 42, 52
system resources 175
manipulating downstream devices 8
mapping
device registers 12
maxBusSet 18
maxLatAllSet 18
maxLatencyArgSet 18
maxLatencyFuncSet 18
mBlk 230
mbuf.h 100
MDIO 78
media access controller, see MAC drivers
media independent interface, see MII
mem32Addr 17, 18
mem32Size 17, 18
memIo32Addr 17, 18
memIo32Size 17, 18
memory technology driver, see MTDs
message signalled interrupt, see MSI
method ID 231
MII 74
MII bus 127
creating 88
deleting 88
interactions with PHY devices 87
lower edge methods 131
lower edge utility routines 133
management 87
relationship to MAC 129
upper edge methods 131
upper edge utility routines 133
miiBus.c 87
miiBus.o 127
miiBusCreate() 78, 88, 129, 135
miiBusDelete() 88
miiBusDevMatch() 128
miiBusMediaAdd() 134
miiBusMediaDefaultSet() 134
miiBusMediaDel() 134
miiBusMediaListGet() 89, 134
miiBusModeGet() 77, 88, 131
miiBusModeSet() 89, 131
miiBusMonitor 77, 88, 127, 131, 135
miiBusRead() 134
miiBusWrite() 135
miiIfName 80
miiIfUnit 80
moduleLib 125
msgLogSet 18
MSI 42, 52
MSI-X 52
MTDs 142
customizing 143
defining
as components 152
in the socket driver file 153
erase routine 152
helper routines 150
initializing the FLFlash structure members 147
non-CFI 145
read routine 151
registering an identification routine 153
supported flash devices 143
write routine 151
writing 146
a map routine 150
read, write, and erase routines 150
the identification routine 147
mtdTable[] 147, 153
multifunction drivers 63
BSP configuration 64
debugging 68
device interconnections 66
driver methods 64
header files 64
initialization 66
interleaved registers 66
location of subordinate devices 68
overview 63
reducing footprint 65
shared resources 67
utility routines 65
muxplexor, see MUX
multiprocessing systems
CPU routing table 42
interrupt controller drivers 36
limitations 60
routing interrupts in 56
MUX 71, 81, 90, 98
attaching to 119
mux2DevConnect() 81
mux2send() 73
muxDevConnect() 81
muxDevLoad() 81, 89, 125
muxDevStart() 81
muxDevStop() 82, 121
muxDevUnload() 82, 121, 125
muxIoctl() 81
muxLib 80, 81
muxSend() 73
muxTkSend() 73
muxTxRestart() 82

N
NAND devices 146
netBufLib 80, 83, 86, 96
netJobQueueId 82
Netperf test suite 120
netPoolCreate() 87
netPoolRelease() 87
netTupleFree() 87
netTupleGet() 87
network device initialization
legacy 91
network drivers 69
see also MAC drivers, PHY drivers
hierarchical END drivers (hEND) 136
interrupts 73
protocols 71
terminology 69
wireless Ethernet drivers 136
network interface drivers, see MAC drivers
network processing tasks 82
networking 83
deferring ISRs 91
interrupt handlers 91
overview 70
ping-of-death 121
protocol impact on drivers 98
receive stall handling 118
setting up a memory pool 96
transmission media 70
NIC_DRV_CTRL 75
nipEndLoad() 75
non-volatile RAM drivers, see NVRAM drivers
notifying the ERF of a new device 194
numSegments 139
NVRAM
  block sizes 140
copying data to 138
stacking instances 141
NVRAM drivers 137
  BSP configuration 139
debugging 141
driver methods 138
header files 139
initialization 140
overview 138
utility routines 140
nvRamSegment 139

O
OHCI 218
Open Host Controller Interface, see OHCI
Open Systems Interconnection (OSI) 70
operating system abstraction layer, see OSAL
operating voltage 158
OS_MALLOC() 219
OSAL 219
other class drivers 225
  BSP configuration 227
debugging 228
driver methods 226
header files 227
initialization 227
overview 225
utility routines 227
overriding
  (*busCfgRead)() 10
  (*busCfgWrite)() 10
  (*vxbDevControl)() 10
  bus controller service routines 9
  register access routines 12

P
pairing a MAC driver with a PHY device 120
parallel drivers 189
see also storage drivers
parameter
definition 231
PARENBR 184
definition 231
PARODD 184
pausing a DMA channel 30
PCI bus
  autoconfiguration 18
  BSP resources 17
  configuration 17
  utility routines for PCI autoconfiguration 19
  utility routines for PCI configuration 19
  pciConfigMechanism 18
  PCMCIA socket drivers 154
pDrvCtrl 75
periodic interrupt timer drivers 200
see also timer drivers
required service routines 205
PHY drivers 69, 87, 127
  BSP configuration 133
debugging 135
device probing and discovery 128
driver methods 130
generic PHY driver support 130
generic TBI driver support 130
genericPhy 127, 128, 130, 132
genericTbiPhy 130
  header files 133
  initialization 135
  lower edge methods 131
  lower edge utility routines 133
  MAC and MII bus relationship 129
  overview 127
  pairing with a MAC driver 120
  upper edge methods 131
  upper edge utility routines 133
  utility routines 133
phyAddr 80, 129
pinCpu 57
PLB 4
definition 231
polling mode
  IPNET-native drivers 126
serial drivers 183
pollInput() 185
pollOutput() 185
powerOnCallback 149
printing isrHandle contents 46
priority 43
priorityTableSize 43
probe 231
probe routine 231
processor local bus, see PLB
programming dynamic vectors 54
programming voltage 159
protocols
   impact on drivers 98

Q
QJOB  92, 111
queueing
   DMA read operations 29
   DMA write operations 29

R
reading and writing device registers 12
reading bus configuration space 5, 6
receive stall handling 118
reducing footprint for multifunction drivers 65
register access 12
   creating a handle for transaction types 13
   handle values for access types 14
   PHY 127
   predefined transaction types 13
   providing a new transaction type 15
registering
   an MTD identification routine 153
   devices and bus types with VxBus 23
   with the ERF 192
registers
   interleaved 66
releasing
   a buffer pool 87
   a DMA channel 27
   a timer 205
removing
   a device from the system 65
   an ISR from isrHandle 45
rerouting interrupts to a specified CPU 38
resident flash array, see RFA
resource
   definition 231
resource drivers 67, 175
   BSP configuration 176
   debugging 177
   driver methods 176
   header files 176
   initialization 177
   overview 175
   utility routines 177
resources, see BSP resources
returning
   a device structure to the pool 24, 66
   a tuple to the pool 87
RFA 154, 155
rfaCardDetected() 156, 158
rfaGetAndClearChangeIndicator() 157, 160
rfaRegister() 155, 158
rfaSetMappingContext() 157, 160
rfaSetWindow() 156, 157, 159
rfaSocketInit() 156, 157, 159
rfaVccOff() 156, 159
rfaVccOn() 156, 158
rfaVppOff() 157, 159
rfaVppOn() 156, 159
rfaWriteProtected() 158, 160
rollcallFuncSet 18
root hub class drivers 218
routines
   (*dmaCancel)() 30
   (*dmaPause)() 30
   (*dmaRead)() 29
   (*dmaReadAndWait)() 29
   (*dmaStatus)() 30
   (*dmaWrite)() 29
   (*dmaWriteAndWait)() 29
   (*timerAllocate)() 205
   (*timerCountGet)() 206
   (*timerCountGet64)() 210
   (*timerDisable)() 207
   (*timerEnable)() 208
   (*timerEnable64)() 209
   (*timerISRSet)() 208
   (*timerRelease)() 205
   (*timerRollOverGet)() 206
   (*timerRollOverGet64)() 209
   (*xf_dump)() 191, 194
   (*xf_ioctl)() 191, 193, 195
   (*xf_strategy)() 191, 194
   bio_done() 191, 194
   callbackInstall() 184
   devInstanceInit2() 83, 85
   devResourceGet() 140
   DMA 28
driverAccessFunc() 15
disCookiePooConfig() 83, 85
disLoad() 89
disPoolCreate() 86
disPoolDestroy() 86
disPoolJumboCreate() 86
disPoolTupleFrie() 86
disPoolTupleGet() 86, 97
erfEventRaise() 191, 194, 195
erfHandlerRegister() 191, 193
erfHandlerUnregister() 191, 193
flDelayMsec() 156
flDontNeedVpp() 150
flNeedVpp() 150
flSetWindowSize() 157
flSocketOff() 158
flWriteProtected() 150
for configuring dynamic vectors 53
hwMemAlloc() 22, 204
ifconfig() 119, 121
in_cksum_skip() 110
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>intCpuUnlock()</td>
<td>51</td>
</tr>
<tr>
<td>intCtlrChainISR()</td>
<td>46, 47</td>
</tr>
<tr>
<td>intCtlrHwConfGet()</td>
<td>44, 45</td>
</tr>
<tr>
<td>intCtlrHwConfShow()</td>
<td>44, 46</td>
</tr>
<tr>
<td>intCtlrISRAdd()</td>
<td>44, 45</td>
</tr>
<tr>
<td>intCtlrISRDisable()</td>
<td>44, 45</td>
</tr>
<tr>
<td>intCtlrISREnable()</td>
<td>44, 45</td>
</tr>
<tr>
<td>intCtlrISRRemove()</td>
<td>44, 45</td>
</tr>
<tr>
<td>intCtlrStrayISR()</td>
<td>46, 47</td>
</tr>
<tr>
<td>intCtlrTableArgGet()</td>
<td>44, 46</td>
</tr>
<tr>
<td>intCtlrTableCreate()</td>
<td>44, 46</td>
</tr>
<tr>
<td>intCtlrTableFlagsGet()</td>
<td>44, 46</td>
</tr>
<tr>
<td>intCtlrTableFlagsSet()</td>
<td>44, 46</td>
</tr>
<tr>
<td>intCtlrTable ISRGet()</td>
<td>44, 46, 47</td>
</tr>
<tr>
<td>intCtlrTableUserSet()</td>
<td>44, 47</td>
</tr>
<tr>
<td>ioctl()</td>
<td>184</td>
</tr>
<tr>
<td>ipcom_drv_eth_init()</td>
<td>119</td>
</tr>
<tr>
<td>isrDeferIsrReroute()</td>
<td>57</td>
</tr>
<tr>
<td>isrRerouteNotify()</td>
<td>57</td>
</tr>
<tr>
<td>jobQueueCreate()</td>
<td>83</td>
</tr>
<tr>
<td>jobQueueInit()</td>
<td>83</td>
</tr>
<tr>
<td>jobQueuePost()</td>
<td>82, 92, 110</td>
</tr>
<tr>
<td>jobQueueStdPost()</td>
<td>82</td>
</tr>
<tr>
<td>ld()</td>
<td>125</td>
</tr>
<tr>
<td>logMsg()</td>
<td>120</td>
</tr>
<tr>
<td>miiBusCreate()</td>
<td>78, 88, 129, 135</td>
</tr>
<tr>
<td>miiBusDelete()</td>
<td>88</td>
</tr>
<tr>
<td>miiBusDevMatch()</td>
<td>128</td>
</tr>
<tr>
<td>miiBusMediaAdd()</td>
<td>134</td>
</tr>
<tr>
<td>miiBusMediaDefaultSet()</td>
<td>134</td>
</tr>
<tr>
<td>miiBusMediaDel()</td>
<td>134</td>
</tr>
<tr>
<td>miiBusMediaListGet()</td>
<td>89, 134</td>
</tr>
<tr>
<td>miiBusModeGet()</td>
<td>77, 88, 131</td>
</tr>
<tr>
<td>miiBusModeSet()</td>
<td>89, 131</td>
</tr>
<tr>
<td>miiBusRead()</td>
<td>134</td>
</tr>
<tr>
<td>miiBusWrite()</td>
<td>135</td>
</tr>
<tr>
<td>moduleLib</td>
<td>125</td>
</tr>
<tr>
<td>MTD helper routines</td>
<td>150</td>
</tr>
<tr>
<td>mux2DevConnect()</td>
<td>81</td>
</tr>
<tr>
<td>mux2send()</td>
<td>73</td>
</tr>
<tr>
<td>muxDevConnect()</td>
<td>81</td>
</tr>
<tr>
<td>muxDevLoad()</td>
<td>81, 89, 125</td>
</tr>
<tr>
<td>muxDevStart()</td>
<td>81</td>
</tr>
<tr>
<td>muxDevStop()</td>
<td>82, 121</td>
</tr>
<tr>
<td>muxDevUnload()</td>
<td>82, 121, 125</td>
</tr>
<tr>
<td>muxIoctl()</td>
<td>81</td>
</tr>
<tr>
<td>muxSend()</td>
<td>73</td>
</tr>
<tr>
<td>muxTkSend()</td>
<td>73</td>
</tr>
<tr>
<td>muxTxRestart()</td>
<td>82</td>
</tr>
<tr>
<td>netPoolCreate()</td>
<td>87</td>
</tr>
<tr>
<td>netPoolRelease()</td>
<td>87</td>
</tr>
<tr>
<td>netTupleFree()</td>
<td>87</td>
</tr>
<tr>
<td>netTupleGet()</td>
<td>87</td>
</tr>
<tr>
<td>nicEndLoad()</td>
<td>75</td>
</tr>
<tr>
<td>OS_MALLOC()</td>
<td>219</td>
</tr>
<tr>
<td>pollInput()</td>
<td>185</td>
</tr>
<tr>
<td>pollOutput()</td>
<td>185</td>
</tr>
<tr>
<td>rfaCardDetected()</td>
<td>156, 158</td>
</tr>
<tr>
<td>rfaGetAndClearChangeIndicator()</td>
<td>157, 160</td>
</tr>
<tr>
<td>rfaRegister()</td>
<td>155, 158</td>
</tr>
<tr>
<td>rfaSetMappingContext()</td>
<td>157, 160</td>
</tr>
<tr>
<td>rfaSetWindow()</td>
<td>156, 157, 159</td>
</tr>
<tr>
<td>rfaSocketInit()</td>
<td>156, 157, 159</td>
</tr>
<tr>
<td>rfaVccOff()</td>
<td>156, 159</td>
</tr>
<tr>
<td>rfaVccOn()</td>
<td>156, 158</td>
</tr>
<tr>
<td>rfaVppOff()</td>
<td>157, 159</td>
</tr>
<tr>
<td>rfaVppOn()</td>
<td>156</td>
</tr>
<tr>
<td>rfwWriteProtected()</td>
<td>158, 160</td>
</tr>
<tr>
<td>setWindow()</td>
<td>157</td>
</tr>
<tr>
<td>stackTxRestartRtn()</td>
<td>82</td>
</tr>
<tr>
<td>sysAuxClkConnect()</td>
<td>213</td>
</tr>
<tr>
<td>sysAuxClkDisable()</td>
<td>213</td>
</tr>
<tr>
<td>sysAuxClkEnable()</td>
<td>213</td>
</tr>
<tr>
<td>sysAuxClkRateGet()</td>
<td>213</td>
</tr>
<tr>
<td>sysAuxClkRateSet()</td>
<td>213</td>
</tr>
<tr>
<td>sysClkConnect()</td>
<td>211</td>
</tr>
<tr>
<td>sysClkDisable()</td>
<td>211</td>
</tr>
<tr>
<td>sysClkEnable()</td>
<td>211</td>
</tr>
<tr>
<td>sysClkRateGet()</td>
<td>211</td>
</tr>
<tr>
<td>sysClkRateSet()</td>
<td>211</td>
</tr>
<tr>
<td>sysTimestamp()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampConnect()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampDisable()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampEnable()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampFreq()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampLock()</td>
<td>214</td>
</tr>
<tr>
<td>sysTimestampPeriod()</td>
<td>214</td>
</tr>
<tr>
<td>txStartup()</td>
<td>184</td>
</tr>
<tr>
<td>unld()</td>
<td>125</td>
</tr>
<tr>
<td>unldLib</td>
<td>125</td>
</tr>
<tr>
<td>usbBusToMem()</td>
<td>220</td>
</tr>
<tr>
<td>usbInit()</td>
<td>222</td>
</tr>
<tr>
<td>usbMemToBus()</td>
<td>220</td>
</tr>
<tr>
<td>usbMemToPci()</td>
<td>220</td>
</tr>
<tr>
<td>usbPciToMem()</td>
<td>220</td>
</tr>
<tr>
<td>usrEnableCpu()</td>
<td>56</td>
</tr>
<tr>
<td>usrNetInit()</td>
<td>119</td>
</tr>
<tr>
<td>vxbBusAnnounce()</td>
<td>20, 23, 66</td>
</tr>
<tr>
<td>vxbDeviceAnnounce()</td>
<td>23, 65</td>
</tr>
<tr>
<td>vxbDeviceDriverRelease()</td>
<td>122</td>
</tr>
<tr>
<td>vxbDevIterate()</td>
<td>226</td>
</tr>
<tr>
<td>vxbDevMethodGet()</td>
<td>122</td>
</tr>
<tr>
<td>vxbDevMethodRun()</td>
<td>122</td>
</tr>
<tr>
<td>vxbDevRemovalAnnounce()</td>
<td>65</td>
</tr>
<tr>
<td>vxbDevStructAlloc()</td>
<td>23, 65</td>
</tr>
<tr>
<td>vxbDevStructFree()</td>
<td>24, 66</td>
</tr>
<tr>
<td>vxb_dmaBufMapIpcomLoad()</td>
<td>87, 95</td>
</tr>
<tr>
<td>vxb_dmaBufMapMblkLoad()</td>
<td>87</td>
</tr>
<tr>
<td>vxb_dmaChanAlloc()</td>
<td>26, 28</td>
</tr>
<tr>
<td>vxbDrvUnregister()</td>
<td>121</td>
</tr>
<tr>
<td>vxbDynIntConnect()</td>
<td>45</td>
</tr>
<tr>
<td>vxbIntConnect()</td>
<td>35, 45, 47, 49, 53</td>
</tr>
<tr>
<td>vxbIntCtrlPinEntryGet()</td>
<td>57</td>
</tr>
<tr>
<td>vxbIntDynaConnect()</td>
<td>53</td>
</tr>
<tr>
<td>vxbIntDynaCtrlInputInit()</td>
<td>48</td>
</tr>
</tbody>
</table>
S

SATA drivers 189
see also storage drivers
SAVE_NVRAM_REGION 144
scatter-gather 95, 97
segments 139
serial ATA drivers, see SATA drivers
serial bitbang 231
serial drivers 179
BSP configuration 182
debugging 187
driver methods 180
header files 181
initialization 182, 185, 186
interaction between serial ports and WDB connection 186
interrupts 185, 186
overview 180
polled versus interrupt-driven mode 183
utility routines 182
WDB 185
service driver 231
setPowerOnCallback 149
setting up a memory pool 96
setWindow() 157
seven layer OSI model 70
shared resources on multifunction chips 67
show routines
debugging MAC drivers 118
intCtrlHwConfShow() 44
printing isrHandle contents 46
vxBusShow() 68
Silicon Image storage driver 192
SIO_AVAIL_MODES_GET 184
SIO_BAUD_GET 184
SIO_BAUD_SET 184
SIO_CALLBACK_GET_TX_CHAR 184
SIO_CALLBACK_PUT_RCV_CHAR 184
SIO_CHAN 180, 183
SIO_CHANNEL_INFO 180, 181
SIO_DRV_FUNCS 183
callbackInstall() 184
ioctl() 184
pollInput() 185
pollOutput() 185
txStartup() 184
SIO_HUP 184
SIO_HW_OPTS_GET 184
SIO_HW_OPTS_SET 184
SIO_MODE_GET 184
SIO_MODE_SET 184, 185
SIO_OPEN 184
diolLib.h 180, 183
diolLibCommon.h 184
slab cache 92
SMP 36
SMP considerations
for interrupt controller drivers 56
timer drivers 215
socket driver file, see sysTfs.c
socket drivers 154
calling the socket registration routines 155
implementing the member function structure 156
multiple drivers 157
PCMCIA 154, 158
required functionality 158
RFA 154
stub file 155
SOCKET_12_VOLTS 159
sockets
address mapping 160
member functions 158
registration 158
windowing 160
spinlocks
using with driverAccessFunc( ) 16
stacking NVRAM instances 141
stackTxRestartRtn( ) 82
stall 231
storage drivers 189
ATA commands 196
BSP configuration 190
debugging 197
driver methods 190
event reporting 195
header files 190
initialization 191
Intel ICH 192
interface with VxWorks file systems 192
overview 189
processing queued work 194
Silicon Image 192
utility routines 191
writing new drivers 196

structures
ATA_RESOURCE 190
ATA_TYPE 190
bio 191, 194
DRV_CTRL 22
END_CAPABILITIES 98, 104, 107, 109
END_MEDIA_LIST 89
END_OBJ 90
FLASH 147, 152
FLSocket 149, 156, 160
for DMA drivers 28
FTL 165
HEND_RX_QUEUE_PARAM 82
intCtlrCpu 42
intrCtlrInputs 40
Ipcom_pkt 72, 92
isrHandle 55
nvRamSegment 139
pDrvCtrl 75
QJOB 92
SIO_CHAN 180, 183
SIO_CHANNEL_INFO 180, 181
SIO_DRV_FUNCS 183
timer drivers 204
vol 156
VXB_ACCESS_LIST 9
VXB_DMA_REQUEST 26
vxbDevRegInfo 21
vxbDmaFuncs 28
vxbDmaResource 28
vxbPciConfig 11
vxbTimerFunctionality 205, 209, 212, 214
VXPI_CTRL_INIT 39, 58
xbd_funcs 191, 193

supporting scatter-gather
IPNET-native drivers 95
M_BLK-style drivers 97
symmetric multiprocessing, see SMP
sysAuxClkConnect( ) 213
sysAuxClkDisable( ) 213
sysAuxClkEnable( ) 213
sysAuxClkRateGet( ) 213
sysAuxClkRateSet( ) 213
sysClkConnect( ) 211
sysClkDisable( ) 211
sysClkEnable( ) 211
sysClkRateGet( ) 211
sysClkRateSet( ) 211
sysHwInit( ) 212, 213, 214
system clock 211
sysTffs.c 144, 145, 146, 152, 153, 154, 155

sysTffsInit( ) 154, 155
sysTimestamp( ) 214
sysTimestampConnect( ) 214
sysTimestampDisable( ) 214
sysTimestampEnable( ) 214
sysTimestampFreq( ) 214
sysTimestampLock( ) 214
sysTimestampPeriod( ) 214

T

tag control information (TCI) 104
tasks 77
  miiBusMonitor 88, 127, 131, 135
tNet0 82, 111, 131
TBI 130
TBI_ID1 130
TBI_ID2 130
terminating an instance 76
terminology
  for network drivers 69
terms
  access routine 229
  advertise 229
  BAR 229
  bus 229
  bus controller 229
  bus discovery 229
  bus match 229
  bus type 229
  child 230
  cluster 230
  descriptor 230
  device 230
downstream 230
driver 230
driver method 230
enumeration 230
instance 230
mBlk 230
method ID 231
parameter 231
parent 231
PLB 231
probe 231
probe routine 231
resource 231
serial bitbang 231
service driver 231
stall 231
upstream 232
testing
  MAC drivers
    loading and unloading 121
    multicast filter test 127
    ping-of-death 121
    polled mode 125
receive error path 126
starting and stopping 121
stress testing 120
with Netperf 120
WTX test 126
tffsConfig.c 147, 154
timer drivers 199
allocating a timer 205
BSP configuration 203
connecting an ISR to a timer 208
data structure layout 204
debugging 215
disabling timer interrupt generation 207
driver methods 200
enabling timer interrupt generation 208
getting
the current value of a timer 206
the maximum return value for a timer 206
header files 203
implementing service routines 205
initialization 203
integrating with VxWorks 211
overview 199
releasing a timer 205
SMP considerations 215
sysHwInit() 212, 213, 214
utility routines 203
VxWorks auxiliary clock 213
VxWorks system clock 211
timers
64-bit 201, 209, 210
connecting an ISR to 208
disabling interrupt generation on 207
enabling interrupt generation on 208
timestamp drivers 200
see also timer drivers
implementing 214
required service routines 205
tNet0 82, 111, 131
tNetTask, see tNet0
transmit-packet-complete interrupt 73
True Flash File System, see TrueFFS
TrueFFS 137, 141
driver development 143
erase units 170
layers
core layer 142
flash translation layer (FTL) 142, 160
MTD layer 142
socket layer 142
overview 142
socket drivers 154
tuple 86
txStartup() 184

U
UHCI 218
Universal Host Controller Interface, see UHCI
universal serial bus, see USB
unld() 125
unldLib 125
upper edge methods 131
upper edge utility routines 133
upstream 232
USB 217
On-The-Go 217
USB drivers 217
address conversion routines 221
address translation 220
BSP configuration 220
debugging 222
driver methods 219
endian conversion for data transfers 220
header files 219
host controller drivers (HCDs) 218
hwconf.c 220
INIT macros 222
initialization 221
example 222
initializing
class drivers 222
USB host controller devices 222
non-VxBus 218
OSAL 219
overview 217
peripheral stack drivers 218
registering with VxBus 221
root hub class 218
USB host stack drivers 218
utility routines 221
VxBus model 218
usbBusToMem() 220
USBD 218
interface 219
usbHst.h 219
usbInit() 222
usbMemToBus() 220
usbMemToPci() 220
usbOsa.h 219
usbPciStub.c 220
usbPciToMem() 220
usrEnableCpu() 56
usrNetInit() 119
utility routines
bus controller drivers 19
DMA drivers 28
interrupt controller drivers 44
MAC drivers 80
multifunction drivers 65
NVRAM drivers 140
other class drivers 227
PCI autoconfiguration 19
Index

V

validating interrupts 120
VBM, see virtual block map
Vcc 158
vector 52
assigned 52
dynamic 52
virtual block map 163
vol 156
Vpp 159
VXB_ACCESS_LIST 9
VXB_BUSDID_MII 20, 128
VXB_BUSDID_PCI 20
VXB_BUSDID_PLB 20
VXB_BUSDID_RAPIDIO 20
VXB_BUSDID_VIRTUAL 20
VXB_DEVID_BUSCTRL 21
VXB_DEVID_DEVICE 21
VXB_DMA_MAP 95
VXB_DMA_REQUEST 26
VXB_DMA_RESOURCE_ID 28
VXB_HANDLE() 15
VXB_HANDLE_IO 14
VXB_HANDLE_MEM 14
VXB_HANDLE_ORDERED 14
VXB_HANDLE_SWAP 14
VXB_HANDLE_WIDTH() 15
VXB_INTCTRL_ISR_CALL() 44, 47, 51
VXB_INTCTRL_PINENTRY_ALLOCATED() 44, 47, 56
VXB_INTCTRL_PINENTRY_ENABLED() 44, 47
VXB_INTCTRL_SPECIFIC_1 46
VXB_INTCTRL_SPECIFIC_2 46
VXB_INTCTRLLIB_LOWVLV_SIZE 55
VXB_INTCTRLLIB_TOPLVLV_SIZE 55
VXB_INTR_DYNAMIC 42, 53
VXB_LEGACY_ACCESS 6, 7, 9
VXB_TIMER_AUTO_RELOAD 201
VXB_TIMER_CAN_INTERRUPT 201
VXB_TIMER_CANNOT_DISABLE 201
VXB_TIMER_CANNOT_MODIFY_ROLLOVER 202
VXB_TIMER_CANNOT_SUPPORT_ALL_FREQS 202
VXB_TIMER_INTERMEDIATE_COUNT 201, 215
VXB_TIMER_SIZE_16 201
VXB_TIMER_SIZE_23 201
VXB_TIMER_SIZE_32 201
VXB_TIMER_SIZE_64 201, 209
VXB_TIMER_STOP_WHILE_READ 201

PCI configuration 19
PHY drivers 133
resource drivers 177
serial drivers 182
storage drivers 191
timer drivers 203
USB drivers 221

vxbAccess.h 9, 14
vxbAuxClkLib 213
vxbAuxClkLib.c 213
vxbBusAnnounce() 20, 23, 66
vxbDeviceAnnounce() 23, 65
vxbDeviceDriverRelease() 122
vxbDevIerate() 226
vxbDevMethodGet() 122
vxbDevMethodRun() 122
vxbDevRegInfo 21
vxbDevRemovalAnnounce() 65
vxbDevStructAlloc() 23, 65
vxbDevStructFree() 24, 66
vxbDmaBufLib 80, 87
vxbDmaBufMapIpcomLoad() 87, 95
vxbDmaBufMapMblkLoad() 87
vxbDmaChanAlloc() 26, 28
vxbDmaDriverLib.h 26
vxbDmaEnd2BufLib 87
vxbDmaFuncs 28
vxbDmaLib 26, 28
vxbDmaLib.h 28
vxbDmaResource 28
vxbDrvUnregister() 121
vxbDynaIntConnect() 45
VxbEnd drivers, see network drivers
vxbEpicIntCrtl.c 34
vxbIntConnect() 35, 45, 49, 53
vxbIntCrtlLib 35, 44, 47, 55
vxbIntCrtlLib.c 46
vxbIntCrtlLib.h 39
vxbIntCrtlPinEntryGet() 57
vxbIntDynaConnect() 53
vxbIntDynaCrtlInputInit() 48
vxbIntDynaVecProgram() 48
vxbIntelIchStorage.c 196
vxbIntrCrtl.h 39
vxbIntToCpuRoute() 56
vxbMsiConnect() 53
vxbNonVol.h 139
vxbNonVolGet() 138
vxbNonVolSet() 138, 141
vxbPci.c 19
vxbPciAutoConfig() 18, 20
vxbPciBusTypeInit() 20, 23
vxbPciConfig 11
vxbPciConfigLibInit() 19
vxbPciMSIProgram() 54
vxbPpciIntCrtl.c 34
vxbRead*() 12
vxbRead16() 12
vxbRead32() 12
vxbRead64() 12
vxbRead8() 12
vxbRegMap() 13
vxbS31xxStorage.c 196
vxbSysClkLib 211
vxbSysClkLib.c 211
vxbTimerFunctionality 205, 209, 212, 214
VxWorks
Device Driver Developer’s Guide, 6.8

VxWorks
Device Driver Developer’s Guide, 6.8

W

WDB 186
kernel initialization 185
serial drivers 185
WDB_COMM_SERIAL 185
WDB_COMM_TYPE 185
Wind River USB, see USB
wireless Ethernet drivers 136
writing
drivers for multifunction devices 63
MTD components 146
MTD identification routine 147
MTD map routine 150
MTD read, write, and erase routines 150
new storage drivers 196
socket drivers for TrueFFS 154
to bus configuration space 7

X

XBD 192
advertising methods 193
creating 192
event types 195
xbd.h 193
xbd_funcs 191, 193
XBD_HARD_EJECT 195
XBD_SOFT_EJECT 195
XBD_STACK_COMPLETE 195
xbdAttach() 191, 193, 194
xbdEventInstantiated 195
xbdEventMediaChanged 195
xbdEventPrimaryInsert 195
xbdEventRemove 195