## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Overview</td>
<td></td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1.1 Technology Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 USB Host Stack Component Overview</td>
<td>2</td>
</tr>
<tr>
<td>Host Controller Drivers</td>
<td>2</td>
</tr>
<tr>
<td>USBD and Class Drivers</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3 USB Peripheral Stack Components Overview</td>
<td>4</td>
</tr>
<tr>
<td>1.1.4 Architecture and BSP Support</td>
<td>6</td>
</tr>
<tr>
<td>1.1.5 Additional Documentation</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Configuration</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Latest Release Information</td>
<td>8</td>
</tr>
<tr>
<td>2  Configuring VxWorks Components</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Configuring USB Host Stack Using Wind River Workbench</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Including USB Host Stack Components</td>
<td>11</td>
</tr>
</tbody>
</table>
2.2.2 Setting USB Host Stack Configuration Parameters ......................... 13
  Configuring Keyboard Parameters .................................................. 13
  Configuring Bulk Mass Storage Device Names .................................. 13
  Configuring CBI Mass Storage Device Names .................................. 14
  Configuring Pegasus Network Device Parameters ............................ 14
2.2.3 Component Dependencies ......................................................... 15
2.3 Configuring USB Peripheral Stack Using Wind River Workbench .......... 15
  2.3.1 Including USB Peripheral Stack Components ............................ 15
2.4 Configuring USB Stack Using the Command Line .............................. 16
  2.4.1 Configuring VxWorks for USB .................................................. 17
     Adding Host Stack Components .................................................. 17
     Adding Peripheral Stack Components ....................................... 19
     Adding the USB Tool Code Exerciser Component ....................... 21
  2.4.2 Managing Dependencies ....................................................... 21
     File System Components ......................................................... 21
     Networking Components ......................................................... 22
     Select Support on USB Keyboard and Mouse Class Drivers .......... 22
     Speaker Demo ................................................................. 23
2.5 Initializing the USB Host Stack Hardware ....................................... 23
  Startup Routines ........................................................................ 23
  Initialization Dependencies ......................................................... 25
  Keyboard, Mouse, Printer, and Speaker Initialization ...................... 25
  Mass Storage Class Device Initialization ..................................... 25
  SCSI-6 Commands ..................................................................... 26
  Communication Class Device Initialization .................................... 27
  USB Audio Demo Initialization ................................................... 27
2.6 Initializing the USB Peripheral Stack Hardware .................................. 28
3 USB Host Drivers ........................................................................... 31
  3.1 Introduction ............................................................................. 31
4 USB Class Drivers ................................................................. 67

4.1 Introduction ............................................................................................................. 67

4.2 Hub Class Driver .................................................................................................... 68
  4.2.1 Registering the Hub Class Driver .......................................................... 68
  4.2.2 Connecting a Device to a Hub ............................................................... 69
  4.2.3 Removing a Device From a Hub ............................................................ 71
  4.2.4 De-registering the Hub Class Driver ..................................................... 71

4.3 Keyboard Driver ..................................................................................................... 72
  4.3.1 SIO Driver Model ..................................................................................... 72
  4.3.2 Initializing the Keyboard Class Driver ................................................. 73
  4.3.3 Registering the Keyboard Class Driver ................................................. 74
  4.3.4 Dynamic Device Attachment ................................................................. 75
  4.3.5 ioctl Routines ............................................................................................. 78
  4.3.6 Data Flow .................................................................................................. 78
  4.3.7 Typematic Repeat ..................................................................................... 79
  4.3.8 Un-initializing the Keyboard Class Driver ........................................... 79

4.4 Mouse Driver ........................................................................................................... 80
  4.4.1 SIO Driver Model ..................................................................................... 80
  4.4.2 Initializing the Mouse Class Driver ....................................................... 80
  4.4.3 Registering the Mouse Class Driver ...................................................... 81
  4.4.4 Dynamic Device Attachment ................................................................. 82
  4.4.5 ioctl Routines ............................................................................................. 83
  4.4.6 Data Flow .................................................................................................. 83
  4.4.7 Un-initializing the Mouse Class Driver ................................................ 83

4.5 Printer Driver .......................................................................................................... 84
  4.5.1 SIO Driver Model ..................................................................................... 84
  4.5.2 Initializing the Printer Driver ..................................................................... 84
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.3</td>
<td>Registering the Printer Driver</td>
<td>85</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Dynamic Device Attachment</td>
<td>86</td>
</tr>
<tr>
<td>4.5.5</td>
<td>ioctl Routines</td>
<td>87</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Data Flow</td>
<td>87</td>
</tr>
<tr>
<td>4.6</td>
<td>Speaker Driver</td>
<td>88</td>
</tr>
<tr>
<td>4.6.1</td>
<td>SEQ_DEV Driver Model</td>
<td>88</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Initializing the Speaker Driver</td>
<td>88</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Registering the Speaker Driver</td>
<td>89</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Dynamic Device Attachment</td>
<td>90</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Recognizing and Handling USB Speakers</td>
<td>91</td>
</tr>
<tr>
<td>4.6.6</td>
<td>ioctl Routines</td>
<td>91</td>
</tr>
<tr>
<td>4.6.7</td>
<td>Data Flow</td>
<td>92</td>
</tr>
<tr>
<td>4.7</td>
<td>Mass Storage Class Driver</td>
<td>93</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Extended Block Device Driver Model</td>
<td>94</td>
</tr>
<tr>
<td>4.7.2</td>
<td>API Routines</td>
<td>96</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Dynamic Attachment</td>
<td>97</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Initialization</td>
<td>98</td>
</tr>
<tr>
<td>4.7.5</td>
<td>Data Flow</td>
<td>98</td>
</tr>
<tr>
<td>4.8</td>
<td>Communication Class Drivers</td>
<td>99</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Ethernet Networking Control Model Driver</td>
<td>99</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Enhanced Network Driver Model</td>
<td>99</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Dynamic Attachment</td>
<td>101</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Initialization</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Interrupt Behavior</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>ioctl Routines</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Data Flow</td>
<td>102</td>
</tr>
</tbody>
</table>
5 USB Peripheral Stack Target Layer Overview ........................................ 103

5.1 Introduction ...................................................................................... 103

5.2 Initializing the Target Layer ................................................................. 105

5.3 Attaching and Detaching a TCD ............................................................. 105

5.3.1 TCD-defined Parameters ................................................................. 106
5.3.2 Detaching a TCD ........................................................................ 107
5.3.3 Target Application Callback Table ................................................. 108

5.4 Enabling and Disabling the TCD .......................................................... 110

5.5 Implementing Target Application Callback Routines ......................... 112

5.5.1 mngmtFunc( ) Callback ................................................................. 113
5.5.2 Clear and Set Callbacks ................................................................ 116
5.5.3 configurationGet( ) Callback ......................................................... 119
5.5.4 configurationSet( ) Callback ......................................................... 119
5.5.5 descriptorGet( ) and descriptorSet( ) Callbacks ......................... 120
5.5.6 interfaceGet( ) Callback ............................................................... 123
5.5.7 interfaceSet( ) Callback ............................................................... 123
5.5.8 statusGet( ) Callback ................................................................ 124
5.5.9 addressSet( ) Callback ............................................................... 125
5.5.10 synchFrameGet() Callback ................................................................. 125
5.5.11 vendorSpecific() Callback ................................................................. 126

5.6 Pipe-Specific Requests ............................................................................ 126
5.6.1 Creating and Destroying the Pipes ....................................................... 127
  Endpoint Descriptor ..................................................................................... 128
  usbTargPipeDestroy() .............................................................................. 130
5.6.2 Transferring and Aborting Data ............................................................. 130
  USB_ERP Structure ..................................................................................... 131
  usbTargTransfer() Routine ........................................................................ 135
  Aborting a Data Transfer ........................................................................... 135
5.6.3 Stalling and Un-stalling the Endpoint ................................................... 135
5.6.4 Handling Default Pipe Requests ........................................................... 136

5.7 Device Control and Status Information .................................................... 137
5.7.1 Getting the Frame Number ................................................................... 137
5.7.2 Resuming the Signal ............................................................................ 137
5.7.3 Setting and Clearing a Device Feature .................................................. 137

5.8 Shutdown Procedures ................................................................................ 138

6 Target Controller Drivers ......................................................................... 141
6.1 Introduction ............................................................................................... 141
6.2 Hardware Adaptation Layer Overview .................................................... 142
6.3 Single Entry Point ...................................................................................... 142
6.4 Target Request Block ................................................................................ 142
6.5 Function Codes .......................................................................................... 144
  6.5.1 Attaching the TCD ............................................................................. 144
  6.5.2 Detaching the TCD ........................................................................... 146
  6.5.3 Enabling and Disabling the TCD ......................................................... 146
6.5.4 Setting the Address ................................................................. 147
6.5.5 Resuming the Signal ................................................................. 147
6.5.6 Setting and Clearing the Device Feature ......................... 147
6.5.7 Getting the Current Frame Number ................................. 148
6.5.8 Assigning the Endpoints ....................................................... 148
6.5.9 Releasing the Endpoints ....................................................... 149
6.5.10 Setting the Endpoint Status .................................................. 149
6.5.11 Getting the Endpoint Status ............................................... 150
6.5.12 Submitting and Canceling ERPs .......................................... 150
6.5.13 Determining Whether the Buffer is Empty ...................... 151
6.5.14 Getting and Clearing Interrupts .......................................... 151
6.5.15 Retrieving an Endpoint-Specific Interrupt ....................... 152
6.5.16 Clearing All Endpoint Interrupts ......................................... 154
6.5.17 Handling Disconnect, Reset, Resume, and Suspend Interrupts 154

7 BSP Porting .................................................................................. 157

7.1 Introduction ................................................................................ 157
7.2 Configuring USB Host Stack Hardware ................................. 158
  7.2.1 Initialization of USB Hardware ........................................ 158
  7.2.2 USB_HCD_BUS_INFO Data Structure .............................. 159
  7.2.3 Locating the Host Controller ........................................... 162
  7.2.4 Porting for PCI Buses ....................................................... 164
  7.2.5 Endian Conversion for PCI Buses ..................................... 166
  7.2.6 Porting for Non-PCI Buses ............................................. 166
        Assignment of Non-PCI Resources .................................. 166
        Endian Conversion ....................................................... 169

7.3 Configuring USB Peripheral Stack Hardware ....................... 169

7.4 Creating BSP-Specific usbPciLib Stub Files ......................... 171
A Runtime Configuration Summary ................................................................. 173

A.1 Introduction ............................................................................................................. 173

A.2 Wind River USB Host Stack Components ............................................................. 174

A.2.1 USB Host Stack ................................................................................................. 174
Locating This Component ........................................................................ 174
Configuration Summary ........................................................................ 174
Dependencies ........................................................................................................ 174

A.2.2 Wind River USB Host Stack Initialization ......................................................... 175
Locating This Component ........................................................................ 175
Configuration Summary ........................................................................ 175
Dependencies ........................................................................................................ 175

A.2.3 Enhanced Host Controller Interface ................................................................. 175
Locating This Component ........................................................................ 175
Dependencies ........................................................................................................ 176

A.2.4 Enhanced Host Controller Interface Initialization .............................................. 176
Locating This Component ........................................................................ 176
Configuration Summary ........................................................................ 176
Dependencies ........................................................................................................ 176

A.2.5 Open Host Controller Interface ........................................................................ 177
Locating This Component ........................................................................ 177
Dependencies ........................................................................................................ 177

A.2.6 Open Host Controller Interface Initialization ..................................................... 177
Locating This Component ........................................................................ 177
Configuration Summary ........................................................................ 178
 Dependencies ........................................................................................................ 178

A.2.7 Universal Host Controller Interface ................................................................. 178
Locating This Component ........................................................................ 178
 Dependencies ........................................................................................................ 178

A.2.8 Universal Host Controller Interface Initialization .............................................. 179
Locating This Component ........................................................................ 179
Configuration Summary ........................................................................ 179
Dependencies ........................................................................................................ 179
A.2.9 USB Keyboard Driver ................................................................. 179
  Locating This Component ......................................................... 179
  Libraries ................................................................................. 180
  Dependencies .......................................................................... 180
A.2.10 USB Keyboard Driver Initialization ........................................ 180
  Locating This Component ......................................................... 180
  Configuration Summary .......................................................... 180
  Dependencies .......................................................................... 180
A.2.11 USB Mouse Driver ................................................................. 181
  Locating This Component ......................................................... 181
  Libraries ................................................................................. 181
  Dependencies .......................................................................... 181
A.2.12 USB Mouse Driver Initialization ............................................. 182
  Locating This Component ......................................................... 182
  Configuration Summary .......................................................... 182
  Dependencies .......................................................................... 182
A.2.13 USB Printer Driver ................................................................. 182
  Locating This Component ......................................................... 182
  Libraries ................................................................................. 183
  Dependencies .......................................................................... 183
A.2.14 USB Printer Driver Initialization ............................................. 183
  Locating This Component ......................................................... 183
  Configuration Summary .......................................................... 183
  Dependencies .......................................................................... 184
A.2.15 USB Speaker Driver ............................................................... 184
  Locating This Component ......................................................... 184
  Libraries ................................................................................. 184
  Dependencies .......................................................................... 184
A.2.16 USB Speaker Driver Initialization ............................................ 184
  Locating This Component ......................................................... 184
  Configuration Summary .......................................................... 185
  Dependencies .......................................................................... 185
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2.17 Bulk Only Mass Storage USB Driver</td>
<td>185</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>185</td>
</tr>
<tr>
<td>Libraries</td>
<td>186</td>
</tr>
<tr>
<td>Dependencies</td>
<td>186</td>
</tr>
<tr>
<td>A.2.18 Bulk Only Mass Storage USB Driver Initialization</td>
<td>186</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>186</td>
</tr>
<tr>
<td>Configuration Summary</td>
<td>186</td>
</tr>
<tr>
<td>Dependencies</td>
<td>187</td>
</tr>
<tr>
<td>A.2.19 USB Mass Storage Driver - Control Bulk Interrupt</td>
<td>187</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>187</td>
</tr>
<tr>
<td>Libraries</td>
<td>187</td>
</tr>
<tr>
<td>Dependencies</td>
<td>188</td>
</tr>
<tr>
<td>A.2.20 USB Mass Storage - CBI Class Driver Initialization</td>
<td>188</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>188</td>
</tr>
<tr>
<td>Configuration Summary</td>
<td>188</td>
</tr>
<tr>
<td>Dependencies</td>
<td>189</td>
</tr>
<tr>
<td>A.2.21 END - Pegasus USB Driver</td>
<td>189</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>189</td>
</tr>
<tr>
<td>Dependencies</td>
<td>189</td>
</tr>
<tr>
<td>A.2.22 End - Pegasus USB IPv4 Driver Initialization</td>
<td>189</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>189</td>
</tr>
<tr>
<td>Configuration Summary</td>
<td>190</td>
</tr>
<tr>
<td>Dependencies</td>
<td>190</td>
</tr>
<tr>
<td>A.2.23 USB Audio Demo</td>
<td>191</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>191</td>
</tr>
<tr>
<td>Configuration Summary</td>
<td>191</td>
</tr>
<tr>
<td>Dependencies</td>
<td>191</td>
</tr>
<tr>
<td>A.3 Wind River USB Peripheral Stack Components</td>
<td>192</td>
</tr>
<tr>
<td>A.3.1 USB Peripheral Stack</td>
<td>192</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>192</td>
</tr>
<tr>
<td>Configuration Summary</td>
<td>192</td>
</tr>
<tr>
<td>A.3.2 NetChip NET2280 USB Peripheral Controller Driver</td>
<td>192</td>
</tr>
<tr>
<td>Locating This Component</td>
<td>192</td>
</tr>
<tr>
<td>Dependencies</td>
<td>193</td>
</tr>
</tbody>
</table>
A.3.3 Philips ISP 1582 USB Peripheral Controller Driver ......................... 193
  Locating This Component .................................................................. 193
  Configuration Summary .................................................................. 193
  Dependencies ................................................................................ 193
A.3.4 PDIUSBD12 USB Peripheral Controller Driver .............................. 194
  Locating This Component ................................................................ 194
  Dependencies ................................................................................ 194
A.3.5 USB Keyboard Emulator ................................................................ 194
  Locating This Component ................................................................ 194
  Libraries ..................................................................................... 195
  Dependencies ................................................................................ 195
A.3.6 Target Keyboard Emulator Driver Initialization ............................. 195
  Locating This Component ................................................................ 195
  Configuration Summary ................................................................ 195
  Dependencies ................................................................................ 195
A.3.7 USB Mass Storage Emulator .......................................................... 196
  Locating This Component ................................................................ 196
  Libraries ..................................................................................... 196
  Dependencies ................................................................................ 196
A.3.8 Target Mass Storage Emulator Driver Initialization ....................... 196
  Locating This Component ................................................................ 196
  Configuration Summary ................................................................ 197
  Dependencies ................................................................................ 197
A.3.9 USB Printer Emulator .................................................................... 197
  Locating This Component ................................................................ 197
  Libraries ..................................................................................... 198
  Dependencies ................................................................................ 198
A.3.10 Target Printer Emulator Driver Initialization ................................. 198
  Locating This Component ................................................................ 198
  Configuration Summary ................................................................ 198
  Dependencies ................................................................................ 198
A.3.11 USB D12 Emulator ....................................................................... 199
  Locating This Component ................................................................ 199
  Dependencies ................................................................................ 199
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4</td>
<td>USB Tool</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>Locating This Component</td>
<td>199</td>
</tr>
<tr>
<td></td>
<td>Configuration Summary</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>Dependencies</td>
<td>200</td>
</tr>
<tr>
<td>B</td>
<td>Glossary</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>Glossary Terms</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>Abbreviations and Acronyms</td>
<td>214</td>
</tr>
<tr>
<td>C</td>
<td>usbTool Code Exerciser Utility Tool</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>Running usbTool from the Shell</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Using the usbTool Execution Sequence</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Testing Applications</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>Testing the Keyboard Application</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>Testing the Printer Application</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Testing the Mass Storage Application</td>
<td>223</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>225</td>
</tr>
</tbody>
</table>
1.1 Introduction

The Wind River USB Programmer’s Guide (this manual) covers both the USB host and peripheral stacks, and documents the following topics:

- It describes the architecture and implementation of the Wind River USB host stack.
- It explains how to use the Wind River USB peripheral stack to create new target applications and target controller drivers.

This manual assumes that you are already familiar with the USB specification, and Wind River’s IDE and VxWorks operating system. Wind River USB has been developed in compliance with the Universal Serial Bus Specification, Revision 2.0, generally referred to in this document as the “USB specification.” Where possible, this manual uses terminology similar to that used in the USB specification so that the correspondence between USB concepts and actions is readily apparent in the software interfaces described.
1.1.1 Technology Overview

The Universal Serial Bus, revision 2.0 (USB 2.0) provides hosts and devices with a versatile channel for communication at low (1.5 Mbps), full (12 Mbps), and high (480 Mbps) data transfer rates, and allows for the following types of transfers:

- control transfers
- bulk transfers
- interrupt transfers
- isochronous transfers

The USB also incorporates provisions for power management and for the dynamic attachment and removal of devices.

This flexibility allows the USB to be used—often concurrently—by a number of different kinds of devices, each requiring its own device driver support. It is desirable that these device drivers be written independent of each other and independent of the implementation of the host computer’s underlying USB host controller interface. Wind River USB meets these requirements, providing a complete set of services to operate the USB and a number of pre-built USB class drivers that handle specific kinds of USB devices.

1.1.2 USB Host Stack Component Overview

Wind River USB is fully compliant with the Universal Serial Bus Revision 2.0.

Host Controller Drivers

USB host controllers are the hardware components responsible for controlling the USB on the board. USB-enabled hardware systems include one or more USB host controllers. Most manufacturers produce USB host controllers that conform to one of three major device specifications. These specifications include the Enhanced Host Controller Interface (EHCI), the Open Host Controller Interface (OHCI), and the Universal Host Controller Interface (UHCI). Wind River provides pre-built
USB host controller drivers (HCDs) for all three specifications as part of the USB host stack product.

USBD and Class Drivers

The USB host stack also provides a USB host driver (USBD). The hardware-independent USBD provides a communication channel between the high layers of the host stack (including the USB class drivers) and the USB. The USBD is responsible for a wide range of tasks including power management, USB bandwidth management, and dynamic attachment and detachment of USB devices. The USBD supports the USB 2.0 specification and is backward compatible with USB 1.0.

The Wind River USB host stack also includes a set of class drivers that are responsible for managing particular types of USB devices. Wind River provides drivers for the following USB device types:
1.1.3 USB Peripheral Stack Components Overview

The Wind River USB peripheral stack is the software component on the USB peripheral that interprets and responds to the commands sent by the USB host. Figure 1-2 shows an overview of the Wind River USB peripheral stack:

![Peripheral Stack Overview Diagram]

- hub
- keyboard
- mouse
- printer
- speaker
- mass storage
- communication

For information on specific tested devices, see your product release notes. For more information on the USB host stack architecture, see 3.2 Architecture Overview, p.31.
At the bottom of the stack is the target controller (TC), the hardware part of the peripheral that connects to the USB. Many manufacturers build target controllers, which vary widely in implementation. For each type of target controller, there is a corresponding Target Controller Driver (TCD). The responsibilities of the TCD are:

- to perform any hardware-specific functionality
- to perform register access (the other layers of the USB peripheral stack are not allowed to perform any register access)
- to implement an entry point used for communication with the upper layers of the USB peripheral stack (the various functions are carried out using different function codes passed to this single entry point)

Above the TCD in the USB peripheral stack is the Hardware Adaptation Layer (HAL). The HAL provides a hardware-independent view of the target controller to higher layers in the stack. This makes it easier to port a target application to new target controller hardware. The HAL makes the implementation of higher layers in the stack (the target layer and target application) completely independent of any hardware-specific peculiarities of the TCD.

In the same way that the HAL is a consistent, abstract mediator for a variety of TCDs, the target layer is a consistent, abstract mediator for a variety of target applications. At run-time, a target application asks the target layer to attach to a TCD on its behalf. The target layer then takes responsibility for routing requests and responses between the TCD and the target application. The target layer can handle multiple TCDs and multiple corresponding target applications.

The functions of the target layer are as follows:

- to initialize and attach the TCD to the target application
- to route requests between the target application and the TCD
- to maintain the default control pipe and manage the transfers with the default control pipe
- to provide interfaces to the target application for creating and deleting pipes to the endpoints
- to provide interfaces for data transfers with the control and the non-control pipes

At the top of the peripheral stack is the target application. A target application responds to USB requests from the host that the TCD routes to the target application through the target layer. For example, when a TCD receives a request to get a USB descriptor, it is the responsibility of the target application to provide the contents of that descriptor.
The interface with the target layer is explained in detail in 5. *USB Peripheral Stack Target Layer Overview* and the interface with the hardware adaptation layer is explained in detail in 6. *Target Controller Drivers*.

1.1.4 Architecture and BSP Support

The Wind River USB host and peripheral stacks support several target architectures. Wind River also provides support for the USB host and peripheral stacks as standard parts for many board support packages (BSPs).

For more information on available architectures and BSPs, see your product release notes or the Wind River Online Support Web site.

1.1.5 Additional Documentation

This manual provides documentation regarding the architecture and implementation of the Wind River USB host stack. Reference pages for USB host stack libraries and subroutines are available in HTML format and can be accessed online from the IDE help menu. You may also want to refer to the Wind River tools documentation and the VxWorks operating system documentation included with your product installation.

Documentation of the USB 2.0 specification is beyond the scope of this manual. For detailed specification information, refer to the following sources:

- USB device class specifications. All USB specifications are available at: [http://www.usb.org/developers/docs](http://www.usb.org/developers/docs)
  - USB Mass Storage Class Overview, Rev. 1.2, June 23, 2003
    [http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf](http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf)
  - USB Mass Storage Bulk Only Transport, Rev. 1.0, September 31, 1999
  - USB Mass Storage Control/Bulk/Interrupt Class (CBI) Transport, Rev. 1.1, June 23, 2003
    [http://www.usb.org/developers/devclass_docs/usb_msc_cbi_1.1.pdf](http://www.usb.org/developers/devclass_docs/usb_msc_cbi_1.1.pdf)
1 Overview

1.1 Introduction

- USB Mass Storage UFI Command Specification, Rev. 1.0, December 14, 1998
- USB Device Class Specification for Printing Devices, Rev. 1.1, January 2000
- Device Class Specification for HID, Rev. 1.11, June 27, 2001
  http://www.usb.org/developers/devclass_docs/HID1_11.pdf

- Host controller specifications:
  - for EHCI:
    http://www.intel.com/technology/usb/ehcispec.htm
  - for OHCI:
    http://www.compaq.com/productinfo/development/openhci.html
  - for UHCI:
    http://www.intel.com/design/USB/UHCI11D.htm
  - for UHCI errata on USB Bandwidth Reclamation, see page 24 in:
    ftp://download.intel.com/design/chipsets/specupdt/29773817.pdf

- Pegasus class specification:
  - for Pegasus II:

For additional information relevant to the USB peripheral stack, refer to the following sources:

- PDIUSBD12 Evaluation Board (PC Kit) User’s Manual, Rev. 2.1, which is included on the floppy disk distributed with the Philips evaluation kit
- Firmware Programming Guide for PDIUSBD12, Version 1.0, which is included on the floppy disk distributed with the Philips evaluation kit
- Universal Serial Bus Specification Rev. 2.0, and USB device class specifications, both of which are available from http://www.usb.org/
- Firmware Programming Guide for NET2280 PCI USB High Speed Peripheral Controller, Rev. 1A available from www.plxtech.com/netchip
- ISP1582/83 Firmware Programming Guide for Philips Hi-Speed Universal Serial Bus Interface Device, Rev. 02 available from www.semiconductors.philips.com
1.2 Configuration

The sample configuration section of the getting started guide for your Platform provides configuration instructions for this component using a default or basic configuration. This book includes a more thorough discussion of each available configuration for the Wind River USB for VxWorks 6. For more detailed project facility, link-time, and run-time configuration information, see 2. Configuring VxWorks Components.

1.3 Latest Release Information

The latest information on this release can be found in the release notes for your Platform. Release notes are shipped with your Platform product and are also available from the Wind River Online Support site:

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

In addition, this site includes links to topics such as known problems, fixed problems, documentation, and patches.

NOTE: Wind River strongly recommends that you visit the Online Support Web site before installing or using this product. The Online Support Web site may include important software patches or other critical information regarding this release.

For information on accessing the Wind River Online Support site, see the Customer Services section of your Platform getting started guide.
2 Configuring VxWorks Components

2.1 Introduction 9

2.2 Configuring USB Host Stack Using Wind River Workbench 10

2.3 Configuring USB Peripheral Stack Using Wind River Workbench 15

2.4 Configuring USB Stack Using the Command Line 16

2.5 Initializing the USB Host Stack Hardware 23

2.6 Initializing the USB Peripheral Stack Hardware 28

2.1 Introduction

You can configure components for VxWorks two different ways. You can use the Wind River Workbench to include components into your VxWorks bootable project, or you can manually edit your BSP configuration files to include or exclude components, and then build VxWorks from the command line.

When choosing a method for configuring and building systems with Wind River USB for VxWorks 6, consider the following criteria:

- Configuring and building from the command line involves editing text files that contain component listings and parameters for initialization, and calling the make utility to build a system image. This process, while possibly faster
than using the project facility, requires that you provide the list of dependent components accurately.

- Configuring and building using the Wind River Workbench provides an easy and accurate method of adding needed components, although the build process can take longer than when using the command line.

For instructions using the Wind River Workbench to configure the USB host and peripheral Stacks, see 2.2 Configuring USB Host Stack Using Wind River Workbench, p.10.

For instructions to edit your BSP configuration files and build VxWorks from the command line, see 2.3 Configuring USB Peripheral Stack Using Wind River Workbench, p.15.

For instructions to build VxWorks from the command line, see 2.4 Configuring USB Stack Using the Command Line, p.16.

For more information on hardware initialization for the USB host and peripheral stacks, see 2.5 Initializing the USB Host Stack Hardware, p.23 and 2.6 Initializing the USB Peripheral Stack Hardware, p.28, respectively.

### 2.2 Configuring USB Host Stack Using Wind River Workbench

This section describes the methods of finding and adding USB components to VxWorks. Locate any of the following components. For a complete description of the USB components discussed in this section, see 2.4.1 Configuring VxWorks for USB, p.17. Once you have located a component, right-click the component name and select Include 'component-name'... to include it in your customized VxWorks image. Dependencies are automatically calculated and you are then prompted to accept the inclusion of dependent components.

After you have included the USB components, you can modify USB configuration parameters in the Properties pane. For more information about the USB configuration parameters, see 2.2.2 Setting USB Host Stack Configuration Parameters, p.13.
2.2 Configuring USB Host Stack Using Wind River Workbench

### 2.2.1 Including USB Host Stack Components

**NOTE:** Last minutes changes may have resulted in differences in the location of the Wind River USB host stack and objects as described here, or as they appear in the IDE. To search for these elements, see the *Wind River Workbench User Interface Reference: Kernel Editor View*, which describes how to use **Find**.

#### Step 1: Bring up the components tree.

To navigate to the component tree in your VxWorks image project:

1. Launch Workbench and open the workspace that contains your VxWorks image project. If you do not have an existing VxWorks image project, consult the *Wind River Workbench User’s Guide* for instructions to create one.

2. Expand your VxWorks image project and right-click **Kernel Configuration** and select **Edit**.

3. In the **Kernel Editor**, select the **Components** tab.

#### Step 2: Locate and include the USB host stack components.

1. Expand the component folders to **hardware > buses > USB Hosts**. At a minimum, you must include:
   - **USB Host Stack** (required) – INCLUDE_USB

   Selecting the **USB Host Stack** component includes support for the USBD.

2. Select at least one host controller driver from the list:
   - **one or more of the following** (required) –
     - **EHCI** – INCLUDE_EHCI
     - **OHCI** – INCLUDE_OHCI
     - **UHCI** – INCLUDE_UHCI

   Selecting the **EHCI**, **OHCI**, or **UHCI** components includes modules for those types of host controllers, respectively. All host controller components require that the **USB Host Stack** component also be selected. More than one host controller can be present on the image at once.

3. To add initialization at system start up, you can include the host stack and host controller driver initialization components. Expand the components folder to **hardware > buses > USB Hosts > USB Host Init** and include the appropriate component(s):
   - **USB Host Stack Init** (optional) – INCLUDE_USB_INIT
4. You can optionally include any of the following USB device components. Expand the components folder to `hardware > peripherals > USB Devices` and include:

- **Keyboard** (optional) – INCLUDE_USB_KEYBOARD
- **Mouse** (optional) – INCLUDE_USB_MOUSE
- **Printer** (optional) – INCLUDE_USB_PRINTER
- **Speaker** (optional) – INCLUDE_USB_SPEAKER
- **Mass Storage - Bulk** (optional) – INCLUDE_USB_MS_BULKONLY
- **Mass Storage - CBI** (optional) – INCLUDE_USB_MS_CBI
- **END - Pegasus** (optional) – INCLUDE_USB_PEGASUS_END

5. To add initialization at startup for peripheral devices, expand the components folder to `hardware > peripherals > USB Devices > USB Device Init` and include the appropriate component(s). These components require that the USB host stack be present on the VxWorks image. To include device initialization at system start up, select any of the USB peripheral device components including the corresponding driver module:

- **Keyboard Init** (optional) – INCLUDE_USB_KEYBOARD_INIT
- **Mouse Init** (optional) – INCLUDE_USB_MOUSE_INIT
- **Printer Init** (optional) – INCLUDE_USB_PRINTER_INIT
- **Speaker Init** (optional) – INCLUDE_USB_SPEAKER_INIT
- **Bulk Mass Storage Init** (optional) – INCLUDE_USB_MS_BULKONLY_INIT
- **CBI Mass Storage Init** (optional) – INCLUDE_USB_MS_CBI_INIT
- **END - Pegasus IPv4 Initialization** (optional) – INCLUDE_USB_PEGASUS_END_INIT

Selecting any of the device initialization components includes the corresponding driver module. These components require that the USB host stack be present on the VxWorks image.
6. You can optionally include support for USB keyboard and mouse class drivers. Expand the components folder to `operating system components > IO system components` and include:

- `select` (optional) – `INCLUDE_SELECT`

For more information, see *Select Support on USB Keyboard and Mouse Class Drivers*, p.22.

**Step 3: Locate and include the USB Testing Tool.**

You can optionally include the USB Testing Tool. Expand the components folder to `hardware > buses` and include:

- `usbTool` (optional) – `INCLUDE_USBTOOL`

The USB Tool performs all necessary USB driver initialization and therefore cannot be used if any of the initialization components (`INCLUDE_XXX_INIT` macros) are included. See *Initialization Dependencies*, p.25 for details.

### 2.2.2 Setting USB Host Stack Configuration Parameters

This section discusses USB host stack runtime configuration parameters that can be set in the IDE by right-clicking the component name and selecting the `Properties` tab.

**Configuring Keyboard Parameters**

- **USB_KBD_QUEUE_SIZE**
  
  Maximum bytes of data that can be stored in the keyboard circular buffer. The default value of `USB_KBD_QUEUE_SIZE` is 8.

**Configuring Bulk Mass Storage Device Names**

- **BULK_DRIVE_NAME**
  
  Indicates the drive name assigned to USB Bulk - Only Device.

- **BULK_MAX_DEVS**
  
  Specifies the maximum number of USB Bulk - Only Devices that can be supported by the driver.
BULK_MAX_DRV_NAME_SZ
Specifies the maximum size of the drive name.

USB_BULK_NON_REMOVABLE_DISK
The new file system prior to every read and write operation does a status check on the mass storage disk to determine whether the media is present or not. For USB flash disks that report themselves as removable media, this may be fine. However, for many USB Mass Storage Disks that consist of Non-Removable Media, the status check prior to every read/write operation can cause unnecessary delays and may impact the performance of the USB Disk. To obtain high performance rates, set USB_BULK_NON_REMOVABLE_DISK to TRUE. The default value is set to FALSE.

Configuring CBI Mass Storage Device Names

CBI_DRIVE_NAME
Indicates the drive name assigned to a the USB CBI device.

UFI_MAX_DEVS
Specifies the maximum number of USB CBI devices supported by the driver.

UFI_MAX_DRV_NAME_SZ
Specifies the maximum size of the CBI drive name.

Configuring Pegasus Network Device Parameters

PEGASUS_IP_ADDRESS
IP address assigned to the USB Pegasus device.

PEGASUS_NET_MASK
USB Pegasus device network mask.

PEGASUS_TARGET_NAME
Target name assigned to the USB Pegasus device.

PEGASUS_MAX_DEVS
Maximum number of supported Pegasus device.

NOTE: For more information about the configuration parameters, including default settings, see A. Runtime Configuration Summary.
2.2.3 Component Dependencies

Certain devices such as EagleTec do not handle mass storage reset properly. For such devices, you can rebuild the source using a compiler flag (macro) so that mass storage reset is not issued to the device. The BULK_RESET_NOT_SUPPORTED macro is undefined by default. For details on defining this flag in the /target/src/drv/usb/makefile and rebuilding the source, see the Getting Started for your platform.

Some components include compiler macros and flags that are defined in the Makefile in the target/src/drv/usb/target directory. This process is documented in the Getting Started for your platform. The USB peripheral stack includes the following configuration options for rebuilding source:

- NET2280_DMA_SUPPORTED – enables DMA transfer for the NET 2280 driver (currently the ISP 1582 driver does not support DMA transfer)

2.3 Configuring USB Peripheral Stack Using Wind River Workbench

2.3.1 Including USB Peripheral Stack Components

Step 1: Locate the USB peripheral stack components.
1. Expand the component folders to hardware > buses. All USB components are located in subfolders of this folder.

Step 2: include the USB peripheral stack components.
1. Expand the hardware > buses folder to USB Target > USB Peripheral Stack. At a minimum, you must include:
   - USB Peripheral Stack (required) – INCLUDE_USB_TARG
2. You must also include a target controller driver. Expand the hardware > buses folder to USB Target > USB Peripheral Controllers and include one of the drivers:
   - one of the following (required) –
     - NetChip NET2280 – INCLUDE_NET2280
– Philips PDIUSBD 12 – INCLUDE_PDIUSBD12
– PhilipsIsp1582 – INCLUDE_PHILIPS1582

3. You can optionally include function drivers (sample emulators). Expand the hardware > buses folder to USB Target > USB Peripheral Stack and include:
   - Keyboard Emulator (optional) – INCLUDE_KBD_EMULATOR
   - Printer Emulator (optional) – INCLUDE_PRN_EMULATOR
   - Mass Storage Emulator (optional) – INCLUDE_MS_EMULATOR
     (this also requires file system components as described in File System Components, p.21)

     The Mass Storage Emulator requires file system components selected from the folder:
     Operating system components > IO system components

4. You can optionally include initialization components for the emulators. Expand the hardware > buses folder to USB Target > Usb Peripheral Init and include the appropriate component(s):
   - Targ Keyboard Emulator Init (optional) – INCLUDE_KBD_EMULATOR_INIT
   - Targ Printer Emulator Init (optional) – INCLUDE_PRN_EMULATOR_INIT
   - Targ Mass Storage Emulator Init (optional) – INCLUDE_MS_EMULATOR_INIT

2.4 Configuring USB Stack Using the Command Line

You can control VxWorks configuration by including or excluding definitions in the target-specific configuration header file config.h. This file is located in the directory target/config/bsp. This file contains definitions that apply only to the specific target, and can also redefine the default definitions, which are found in configAll.h, that are inappropriate for the particular target. For example, if a target cannot access a device controller at the default I/O address defined in configAll.h because of addressing limitations, the address can be redefined in config.h.
2 Configuring VxWorks Components

2.4 Configuring USB Stack Using the Command Line

NOTE: Do not edit configAll.h.

The config.h header file includes definitions for the following parameters:

- default boot parameter string for boot ROMs
- interrupt vectors for system clock and parity errors
- device controller I/O addresses, interrupt vectors, and interrupt levels
- shared memory network parameters
- miscellaneous memory addresses and constants

2.4.1 Configuring VxWorks for USB

Adding Host Stack Components

USB Host Driver (USBD) Component

To configure the USBD, define the following components.

#include<USB

The USB Host driver component, required by all systems using USB.

#include<USB_INIT

Initialization component for the USB Host driver (USBD).

Host Controller Driver (HCD) Components

In addition to a host driver, you must include at least one host controller driver in your basic USB configuration. You can select more than one, if needed.

#include<EHCl

The EHCI host controller driver.

#include<OHCl

The OHCI host controller driver.

#include<UHCl

The UHCI host controller driver.

Host Controller Driver Initialization Components

If you want to include initialization at startup for any of the host controller drivers, add the corresponding initialization component. The initialization components are not compatible with the USB Tool component.
INCLUDE_EHCI_INIT
Initialization component for the EHCI driver.

INCLUDE_OHCI_INIT
Initialization component for the OHCI driver.

INCLUDE_UHCI_INIT
Initialization component for the EHCI driver.

USB Driver Components

If you plan to connect a particular USB device to your host system, add the appropriate class driver for that device type.

INCLUDE_USB_KEYBOARD
USB keyboard class driver.

INCLUDE_USB_MOUSE
USB mouse class driver.

INCLUDE_USB_PRINTER
USB printer class driver.

INCLUDE_USB_SPEAKER
USB speaker class driver.

USB Driver Initialization Components

If you want to include initialization at startup for any of the USB class drivers, add the corresponding initialization component. The initialization components are not compatible with the USB Tool component.

INCLUDE_USB_KEYBOARD_INIT
Initialization component for the USB keyboard class driver.

INCLUDE_USB_MOUSE_INIT
Initialization component for the USB mouse class driver.

INCLUDE_USB_PRINTER_INIT
Initialization component for the USB printer class driver.

INCLUDE_USB_SPEAKER_INIT
Initialization component for the USB speaker class driver.

Mass Storage Components

If your system is configured with a mass storage class driver, you must also include support for *File System Components*, p.21.
2 Configuring VxWorks Components

2.4 Configuring USB Stack Using the Command Line

INCLUDE_USB_MS_BULKONLY
USB mass storage bulk-only class driver.

INCLUDE_USB_MS_CBI
USB mass storage CBI class driver.

Mass Storage Device Initialization Components

If you want to include initialization at startup for any of the USB mass storage drivers, add the corresponding initialization component. The initialization components are not compatible with the USB Tool component.

IINCLUDE_USB_MS_CBI_INIT
Initialization component for USB mass storage CBI class driver.

INCLUDE_USB_MS_BULKONLY_INIT
Initialization component for USB mass storage bulk-only class driver.

Pegasus Network Device Components

In order to include USB Pegasus END driver initialization, you must also include network initialization. See Networking Components, p.22.

INCLUDE_USB_PEGASUS_END
USB Pegasus communication class driver.

INCLUDE_USB_PEGASUS_END_INIT
Initialization component for USB Pegasus communication class driver.

The Pegasus default configuration parameters are:

#define PEGASUS_IP_ADDRESS {"90.0.1.3"}
#define PEGASUS_NET_MASK {0xffffff00}
#define PEGASUS_TARGET_NAME {"usbTarg0"}
#define PEGASUS_MAX_DEVS 1

See also Configuring Pegasus Network Device Parameters, p.14 and A.2.21 END - Pegasus USB Driver, p.189.

NOTE: For information on setting configuration parameters for the USB components, see 2.2.2 Setting USB Host Stack Configuration Parameters, p.13.

Adding Peripheral Stack Components

Wind River USB Peripheral Stack Core Component

To configure the USB Peripheral Stack, define the following components:
INCLUDE_USB_TARG
This is the core component of USB 2.0 Peripheral Stack and must be included for all systems.

INCLUDE_USB_TARG_INIT
Initialization component for the core component for USB 2.0 Peripheral Stack.

Target Controller Driver (TCD) Components
Include one of the following components to include the target controller driver.

INCLUDE_NET2280
This macro includes the Netchip NET2280 target controller driver.

INCLUDE_PDIUSBD12
This macro includes the Philips PDIUSBD 12 target controller driver.

INCLUDE_PHILIPS1582
This macro includes the Philips ISP1582 target controller driver.

USB Peripheral Stack Functionality Components
Include one or more of the following components in the configuration.

INCLUDE_KBD_EMULATOR
This macro includes the keyboard emulator.

INCLUDE_KBD_EMULATOR_INIT
This macro initializes the keyboard emulator.

INCLUDE_PRN_EMULATOR
This macro includes the printer emulator.

INCLUDE_PRN_EMULATOR_INIT
This macro initializes the printer emulator.

INCLUDE_MS_EMULATOR
This macro includes the mass storage emulator.

INCLUDE_MS_EMULATOR_INIT
This macro initializes the mass storage emulator.
Adding the USB Tool Code Exerciser Component

USB Tool Component

`INCLUDE_USBTOOL`
Code exerciser, which performs all necessary USB driver initialization and can be used for debugging purposes. If `INCLUDE_USBTOOL` is included, all initialization components (`INCLUDE_XXX_INIT`) must be undefined.

2.4.2 Managing Dependencies

This section discusses the general component dependencies for the USB host stack. For more information about the components discussed in this section, see the *VxWorks Programmer’s Guide* (for VxWorks 5.5 users) or the *VxWorks Kernel Programmer’s Guide* (for VxWorks 6.1 users).

For component dependencies are specified in the component descriptor file, see *A. Runtime Configuration Summary*.

File System Components

If your system is configured with mass storage class or function drivers, you must also include support for a file system. You can use dosFs, HRFS (Highly Reliable File System), and rawFs with the USB host stack. The dosFs and HRFS file systems can exist simultaneously.

To use a particular file system, the disk must be formatted under the chosen file system, and you must include the appropriate components required for that file system and with the event framework. Following is a list of examples, some or all of which might be required, depending upon your particular configuration:

```
#define INCLUDE_DOSFS
#define INCLUDE_DOSFS_MAIN
#define INCLUDE_DOSFS_CHKDSK
#define INCLUDE_DOSFS_FMT
#define INCLUDE_FS_MONITOR
#define INCLUDE_ERF
#define INCLUDE_XBD
#define INCLUDE_DEVICE_MANAGER
#define INCLUDE_XBD_PART_LIB
```
In addition, there are other file system components that are not required, but which may be useful. These components add support for the basic functionality that one needs to use a file system, such as the commands `ls`, `cd`, `copy`, and so on.

For details, see the file system chapters in the *VxWorks Application Programmer’s Guide* and the *VxWorks Kernel Programmer’s Guide* for your platform.

### Networking Components

In order to include USB Pegasus END driver initialization, you must also include network initialization. In addition, you may need to increase the `IP_MAX_UNITS` value, depending on your system requirements. For example, in a system that initializes an FEI Ethernet interface and a USB Pegasus Ethernet interface, the `IP_MAX_UNITS` must be at least 2 (in VxWorks 5.5, the default value is 1; for VxWorks 6.0, the default value is 4). If you wish to test the Ethernet device, you must also include the network show routines component.

The network component `INCLUDE_IFCONFIG` needs to be included for the Pegasus interface support due to network stack changes. This is not required in the USB Host Stack, 2.1.

### Select Support on USB Keyboard and Mouse Class Drivers

The USB keyboard and mouse class drivers provide support for the select feature, which allows multiple tasks to pend on the driver while waiting for I/O requests. Once the devices are ready to send data, the driver notifies the tasks pending on them. To support the select feature, the keyboard and mouse drivers must be set to interrupt mode. Do this by issuing an `ioctl` request, with request option as `SIO_MODE_INT`.

**NOTE:** Keyboard and mouse class drivers are set to interrupt mode by default.

To configure VxWorks to support the select feature for USB keyboard and mouse class drivers, add the following line to your `config.h` file:

```
#define INCLUDE_SELECT
```

For more information on the `select()` API call, see the reference entry.
2.5 Initializing the USB Host Stack Hardware

This section provides initialization instructions for all USB host stack sub-components including host controller initialization, device initialization, and USB demo initialization.

Startup Routines

This section provides a detailed description of USB initialization process.

The USB host stack initialization process includes two parts:

- initializing the USBD (USB Host Driver) component
- attaching the EHCI, OHCI, and UHCI host controllers

The USB host stack includes initialization configlettes for the USBD (host stack) and for EHCI, OHCI, and UHCI host controllers. These initialization configlettes are present in the installDir/vxworks-6.x/target/config/comps/src directory.

USBD Initialization

To initialize the USB host driver (USBD), call the routine `usbInit()` defined in the file installDir/vxworks-6.x/target/config/comps/src/usrUsbInit.c.

Figure 2-1 shows the responsibilities of the `usbInit()` configlette routine and the initialization process.
Attaching the EHCI, OHCI, and UHCI Host Controllers

Once the USBD is initialized, next it is required to attach the host controllers. The following configlette routines are available to attach different type of host controllers.

usrUsbHcdUhciAttach()
Configlette routine to attach a UHCI host controller. Defined in the file usrUsbHcdUhcdInit.c.

usrUsbHcdEhciAttach()
Configlette routine to attach an EHCI host controller. Defined in the file usrUsbHcdUhcdInit.c.
2.5 Initializing the USB Host Stack Hardware

usrUsbHcdOhciAttach()
Configlette routine to attach an OHCI host controller. Defined in the file
usrUsbHcdUhcInit.c.

NOTE: If the initialization components are included, the configlette routines are
called to initialize the USBD and host controllers during the boot-up sequence
itself.

Initialization Dependencies

The USB host stack includes initialization components for all of the USB modules.

⚠️ CAUTION: When you use the initialization components, usbTool cannot be used.
The components are derived from usbTool itself, and cause compile-time errors
when used concurrently with usbTool.

Keyboard, Mouse, Printer, and Speaker Initialization

The keyboard, mouse, printer, and speaker drivers contain initialization routines
that install standard open, close, read, write, and ioctl routines into the I/O system
driver table. This allows an application to call these drivers using standard
VxWorks system calls.

For example, an application might want to monitor a keyboard’s input. First, the
application opens the keyboard with the system call to open():

```c
fileDescr = open ("/usbkb/0", 2, 0);
```

The application can now call the system’s read() routine in a loop:

```c
while (read (fileDescr, &inChar, 1) != 1);
```

These operations can be used for the mouse, printer, and speaker drivers as well.

Mass Storage Class Device Initialization

The bulk-only and CBI mass storage class driver configlettes install standard
routines into a file system. As with the mouse, keyboard, speaker, and printer
drivers, these routines allow an application to make standard VxWorks system
calls, such as copy, rm, and format (depending on which file system is attached),
to access the mass storage class device.
After a USB block device is created by means of `usbMSCBlkDevCreate()`, a file system can be attached to the device as follows:

```c
/* attach DOS file system to the USB drive */
pBulkDosVol = dosFsDevCreate ("/usbDr0", pBulkBlkDev, MAX_NO_FILES, 0);
```

Now calls such as `copy()` can refer to the device as `usbDr0`. In the following code fragment, the file `readme.txt` is copied from a host to the USB drive `usbDr0`.

```c
copy ("host:/readme.txt", "/usbDr0/readme.txt");
```

A file system can be attached to the device if the appropriate file system component is included, the device has been attached to the XBD, and the insertion event has been sent to the Event Reporting Framework. The following code, which is implemented in `usrUsbBulkDevInit.c`, can be used as an example:

```c
If (xbdAttach (pBulkXbdDev, &usbBulkXbdLunFuncs,
    pBulkDevice->usbBulkDrvName[lun], pBulkXbdDev->xbd_blocksize,
    pBulkXbdDev->xbd_nblocks, &retVal) == OK)
{
    pBulkDevLun->usbBulkXbdFsRemoved = FALSE;
    erfEventRaise (xbdEventCategory, xbdEventPrimaryInsert,
        ERF_ASYNC_PROC, (void *)retVal, NULL);
}
```

## SCSI-6 Commands

In internal testing, Wind River has found that one supported drive, the M-Systems FlashOnKey device, does not support SCSI-6 commands. This may also be the case for some non-supported drives. Therefore, the bulk-only driver supports both SCSI-6 and SCSI-10 read/write commands. The user must configure the driver to use the appropriate command for communicating with the device.

The fifth parameter of `usbMSCBlkDevCreate()` sets the SCSI transfer mode. It can take either of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>USB_SCSI_FLAG_READ_WRITE10</code></td>
<td>Use SCSI read/write 10.</td>
</tr>
<tr>
<td><code>USB_SCSI_FLAG_READ_WRITE6</code></td>
<td>Use SCSI read/write 6.</td>
</tr>
</tbody>
</table>
2 Configuring VxWorks Components
2.5 Initializing the USB Host Stack Hardware

Communication Class Device Initialization

The USB host stack includes a configlette, called `usrUsbPegasusEndInit.c`, to initialize the Pegasus communication class driver. Upon device insertion, these routines connect a Pegasus device to the network stack, attaching an IP address to the device.

The IP address (`PEGASUS_IP_ADDRESS`), target name (`PEGASUS_TARGET_NAME`), a net mask (`PEGASUS_NET_MASK`), and maximum Pegasus devices (`PEGASUS_MAX_DEVS`) are all user-definable parameters of the component and must be set before communication with the Pegasus device can occur. Since the current version of Pegasus driver supports multiple Pegasus devices, customers can increase the `PEGASUS_MAX_DEVS` and expand the `PEGASUS_IP_ADDRESS`, `PEGASUS_NET_MASK`, and `PEGASUS_TARGET_NAME` arrays for the multiple Pegasus interface configuration. For more information, see Configuring Pegasus Network Device Parameters, p.14, Pegasus Network Device Components, p.19, and A. Runtime Configuration Summary.

USB Audio Demo Initialization

The USB host stack includes a sample application called `usbAudio`, which demonstrates how to use the USB host stack and the speaker class driver. This application was designed to run on an Intel Architecture (`pcPentium`) machine that contains both a host controller (EHCI, OHCI, or UHCI) and an ATA hard drive containing `.wav` files.

The `usbAudio` program operates with any of the listed supported speakers. For a complete list of supported speakers, see your Platform release notes.

**NOTE:** When including USB Audio Demo components (`INCLUDE_AUDIO_DEMO`), the speaker initialization component (`INCLUDE_USB_SPEAKER_INIT`) should not be included. The speaker is initialized by the USB Audio Demo itself.
2.6 Initializing the USB Peripheral Stack Hardware

The USB peripheral stack contains configlette routines that are used to locate and configure resources required for initializing the USB peripheral hardware. These configlette routines are defined in `/target/config/comp/src/usrUsbTargPciInit.c`. For more information, see 5.4 Enabling and Disabling the TCD, p.110.

The responsibilities of this configlette layer are:

- Depending on the type of controller, make a call to locate the controller. For example, for a PCI-based target controller, call PCI class files to locate the hardware.
- Determine the base address and the interrupt line for the hardware. This can be hard-coded or can be obtained from the PCI configuration header.
- Determine any other hardware-specific features.

Example 2-1 NET2280 Configlette Routine

The code fragment below illustrates the implementation of the configlette routine for the NET2280 PCI-based peripheral controller:

```c
void sys2NET2280PciInit (void)
{
    int PCIBusNumber; /* PCI bus number */
    int PCIDeviceNumber; /* PCI device number */
    int PCIFunctionNumber; /* PCI function number */
    UINT32 bStatus; /* status */
    PCI_CFG_HEADER pciCfgHdr; /* configuration header */
    UINT8 i = 0;

    /* Find the device */
    bStatus = USB_PCI_FIND_DEVICE (NET2280_VENDOR_ID, 
                                  NET2280_DEVICE_ID, nDeviceIndex, &PCIBusNumber, 
                                  &PCIDeviceNumber, &PCIFunctionNumber);

    /* Check whether the NET2280 Controller was found */
    if (bStatus != OK )
    {
        /* No NET2280 Device found */
        printf ("pciFindDevice returned error \n ");
        return ;
    }

    /* Get the configuration header */
    usbPciConfigHeaderGet (PCIBusNumber, PCIDeviceNumber, 
                           PCIFunctionNumber, &pciCfgHdr);

    /* Obtain the base address */
```
BADDR_NET2280[i] = pciCfgHdr.baseReg[i];

/* Obtain the interrupt line */

IRQ_NET2280 = pciCfgHdr.intLine - INT_NUM_IRQ0;
return;
}

NOTE: The hardware configuration routine calls BSP routines for USB. The BSP interface for USB is defined in the file /target/config/BSP/usbPciStub.c. For the interface definitions, see 7. BSP Porting.
3.1 Introduction

This chapter provides a detailed overview of the USB host stack architecture as well as detailed usage information regarding the USB host driver (USBD) and the USB host controller drivers. Information on USB class drivers is available in 4. USB Class Drivers.

3.2 Architecture Overview

Figure 3-1 presents a simplified overview of the USB host stack architecture.
Figure 3-1  **USB 2.0 Host Stack**

- cmdParser
- New Class Drivers
- usbMouseLib
- usbKeyboardLib
- Existing USB1.1 Class Drivers
- USB 2.0 Hub Class Driver
- usbTool USB Exerciser Utility
- USB 2.0 Translation Layer

**OSAL**

**USBD 2.0**
- Initialization
- Registrar Interface
- Device Manager
- USB Request Interface

**USB 2.0 Host Controller Drivers**
- UHCI HCD
- OHCI HCD
- EHCI HCD

**PCI Host Controllers**
**Non-PCI Host Controllers**
At the bottom of the stack is the USB host controller (USB HC), the piece of hardware in the host system that carries out the USB operations. Currently, there are three major families of USB host controllers on the market, those supporting the Enhanced Host Controller Interface (EHCI) which supports high-speed transfers; those supporting the Open Host Controller Interface (OHCI) designed by Microsoft, Compaq, and National Semiconductor; and those supporting the Universal Host Controller Interface (UHCI) originally put forth by Intel. A number of hardware manufacturers have built USB HCs around one or more of these specifications.

Host Controller Drivers and USBD

For each type of host controller there is a single, hardware-dependent USB host controller driver (HCD). Wind River provides the source for three pre-built drivers in the following directories:

- target/src/usb2/hcd/ehcd for EHCI HCs
- target/src/usb2/hcd/ohcd for OHCI HCs
- target/src/usb2/hcd/uhcd for UHCI HCs

The interface between the USB host driver (USBD) and the HCD allows for each HCD to control one or more underlying HCs. Also, Wind River’s USBD is capable of connecting to multiple USB HCDs simultaneously. These design features allow you to build a range of complex USB systems.

The USBD is the hardware-independent module above the HCD(s). The USBD manages each USB connected to the host and provides the path through which higher layers communicate with the USB.

Among its responsibilities, the USBD (USB Driver) implements the USB protocol as explained in USB 2.0 Specification. It handles all the standard requests and routes other types of requests to the class drivers. Hub functionality is critical to the proper operation of the USB, so Wind River USBD hub functionality is handled transparently by the Hub Class Driver. The responsibilities of the Hub Class Driver include handling device connects and disconnects and power management.

In the Wind River USB host stack, the USBD 2.0 and the hub class modules are implemented in the following directory structures, respectively:

- target/src/usb2/usbd
- target/src/usb2/hub
Class Drivers

Figure 3-1 shows the USB client module at the top of the stack. USB class drivers are typical examples of client modules. USB class drivers are responsible for managing individual types of devices that can be connected to the USB; they rely on the USBD to provide the communication path to the individual devices. Applications, diagnostics, and test programs are other examples of client modules that rely on the USBD to communicate with USB devices. For example, Wind River provides the test application/module **usbTool**, which gives you interactive control over the USB bus and devices.

In the Wind River USB host stack, the class driver modules are implemented at the following directory structure:

```
target/src/drv/usb
```

Host Module Roadmap

The diagram in Figure 3-1 illustrates the functional relationships between the modules that comprise the USB host stack. Each module is further described in this section; for additional information on USB-related libraries and subroutines, see the associated API reference entry.

**usbTool**

This module is a test application that gives you interactive control of the USB host stack. The **usbTool** utility exports a single entry point, **usbTool()**, that invokes a command-line-driven interactive environment in which the operator can initialize components of the USB host stack, interrogate each USB device connected to the system, send USB commands to individual USB devices, and test other elements of the USB stack. The **usbTool** test application is most useful during development and testing and does not need to be included in your final product.

The **usbTool** utility resides in **target/config/comps/src** and is called **usrUsbTool.c**.

**NOTE:** The **usbTool** module relies internally on static data; do not run multiple instances of **usbTool** simultaneously.

**cmdParser**

This module provides the generalized command-line parsing routines used by **usbTool**. The **cmdParser** module needs to be present only when **usbTool** is being used.
usbKeyboardLib
  This module is the USB class driver for USB keyboard devices; for more information, see 4.3 Keyboard Driver, p.72.

usbMouseLib
  This module is the USB class driver for USB mouse devices; for more information, see 4.4 Mouse Driver, p.80.

usbPrinterLib
  This module is the USB class driver for USB printer devices; for more information, see 4.5 Printer Driver, p.84.

usbSpeakerLib
  This module is the USB class driver for USB speaker devices; for more information, see 4.6 Speaker Driver, p.88.

usbBulkDevLib and
usbCbiUfiDevLib
  These modules are the USB class drivers for the USB mass storage class devices; for more information, see 4.7 Mass Storage Class Driver, p.93.

usbPegasusEndLib
  These modules are the USB class drivers for the USB Ethernet networking control model communication class devices; for more information, see 4.8.1 Ethernet Networking Control Model Driver, p.99.

USB 2.0 Hub Class driver
  The USB 2.0 hub class driver is responsible for managing device connection, disconnection, and the power management functionality.

USB 2.0 Translation Layer
  This module contains the functionality to preserve the backward compatibility of the Wind River-supplied class drivers (keyboard, mouse, mass storage drivers, and so forth) from the USB host stack, 1.1 (USB 1.0) with the updated USB host stack, (USB 2.0) interface. That is, existing USB class drivers interface with the USB 2.0-based host stack through the USB 2.0 translation layer.

USB 2.0 USBD Layer
  This layer contains the functionality of the USB host stack USBD and is composed of various modules: the initialization module is responsible for initializing global data structures; the registrar provides an interface for class drivers and host controller drivers to register with the USBD; the device manager provides interfaces for the hub class driver to manage device connection, disconnection, suspend, and resume; and the USB request interface provides an interface for class drivers to issue standard USB requests or vendor-specific requests to the devices.
USB 2.0 Host Controller Drivers
These modules drive the USB host controllers (EHCI, OHCI, and UHCI) attached to the system.

usbPciLib
The module usbPciLib exists conceptually; however, it is available in the form of BSP-dependent files of the format target/config/BSP/usbPciStub.c. This module provides HCDs with an abstracted view of the underlying hardware (both PCI and non-PCI) and software in the system.

The usbPciLib stub files shield the HCDs from variation among BSPs. This allows HCDs to be written independent of the underlying BSP, and promotes portability of HCDs across BSP platforms and hardware systems.

For details, see 7. BSP Porting.

OSAL
This component provides an abstracted and simplified view of the VxWorks operating system services to the host stack. OSAL includes routines that manage tasks, mutexes, implied semaphores, memory allocation, and system time.

3.3 The USB Host Driver (USBD)
This section describes initialization, client registration, dynamic attachment registration, device configuration, and data transfers. It also discusses key features of the USBD internal design. This section is divided into two subsections. The first subsection describes the USB 2.0 USB host driver (USBD 2.0) and the second subsection describes compatibility with the USB 1.0 USBD (USBD 1.1).

3.3.1 USBD 2.0
This section describes the USB host driver (USBD) for USB 2.0.
Initializing the USBD

When using the USBD 2.0 interface, initializing the USBD is a three-step process. First, the USBD entry point, `usbdInit()`, must be called at least once. The `usbdInit()` routine initializes internal USBD data structures. In a given system, it is acceptable to call `usbdInit()` once (for example, during the boot sequence) or to call it many times (as during the initialization of each USBD client). The second step is to call `usbHubInit()` to initialize the hub class driver. The final step of initializing the USBD is to attach at least one of the HCDs by calling the appropriate `usbHcdInit()` routine, where `Hcd` is either `Ehcd` (EHCI HCD), `Ohci` (OHCI HCD), or `Uhcd` (UHCI HCD).

Normally, the attachment of HCDs to the USBD occurs during the VxWorks boot sequence. However, the USBD is designed to allow HCDs to be attached (and removed) at run-time. This flexibility allows the USBD to support systems in which the USB host controllers can be “hotswapped” without restarting the system.

Order of Initialization

The `usbdInit()` routine must be called before all other USBD routines, including `usbHubInit()`. However, not all USBD clients need to call `usbdInit()` before one or more HCDs have been attached to the USBD. Either of the following initialization sequence scenarios is acceptable:

**Scenario #1: Traditional Boot Time Initialization**

The order of initialization is as follows:

1. Call `usb2PciInit()`.
2. Call `usbdInit()`.
3. Call `usbHubInit()`.
4. Call `usbOhciInit()`, `usbEhcdInit()` or `usbUhcdInit()` to attach the respective HCD.
5. Call the USB class driver initialization entry point.

**Scenario #2: Hotswap-Driven Initialization**

1. Create a VxWorks image that includes the following components:
   - USB Host Stack
One or more USB Host Controllers

usbTool

**NOTE:** Refer to *2.2 Configuring USB Host Stack Using Wind River Workbench*, p.10 for how to include these components.

2. The hotswap code should call the function `usb2PciInit()`.
3. The hotswap code should call the function `usbInit()`.
4. The hotswap code should call the function `usbOhciInit()`, `usbEhciInit()`, or `usbUhciInit()` to attach the respective controller.
5. Call the entry point of the respective class driver.

Similarly, calls to `usbOhciExit()`, `usbEhcdExit()`, and `usbUhcdExit()` can be made at any time to detach the respective HCDs from the USBD.

**Bus Tasks**

For each host controller attached to the USBD, the hub class driver spawns a bus task responsible for monitoring bus events, such as the attachment and removal of devices. These tasks are normally “dormant” —that is, consuming no CPU time—and they typically wake up only when a USB hub reports a change on one of its ports.

Each USBD bus task has the VxWorks task name `BusM A`, `BusM B`, and so forth.

**Registering Client Modules**

The client module and the USBD layer share the `USBHST_DEVICE_DRIVER` data structure, defined in the file `target/h/usb2/usbHst.h` as follows:

```c
/*
 * This structure is used to store the pointers to entry points and class
 * driver information
 */

typedef struct usbhst_device_driver
{
    /* Vendor Specific or class specific flag */
    BOOL bFlagVendorSpecific;
```
3 USB Host Drivers

3.3 The USB Host Driver (USBD)

In order to communicate with USB devices, the client modules must register this data structure with the USBD by calling `usbHstDriverRegister()`. When a client registers with the USBD, the USBD allocates per-client data structures that are later used to track all requests made by that client.

There are two types of USB Devices:

```c
/* Vendor ID (if vendor) or Class Code */
UINT16 uVendorIDorClass;

/* Dev ID (if vendor) or SubClass Code */
UINT16 uProductIDorSubClass;

/* DevRel num (if vendor) or Protocol code */
UINT16 uBCDUSBorProtocol;

/* Function registered as to be called when
   * a matching interface/device is connected
   */

USBHST_STATUS (*addDevice) (UINT32 hDevice,
                            UINT8 uInterfaceNumber,
                            UINT8 uSpeed,
                            void **pDriverData);

/* Function registered as to be called when
   * a matching interface/device is disconnected
   */

void (*removeDevice)(UINT32 hDevice,
                      void * pDriverData);

/* Function registered as to be called when
   * a matching interface/device is suspended
   */

void (*suspendDevice)(UINT32 hDevice,
                      void * pDriverData);

/* Function registered as to be called when
   * a matching interface/device is resumed
   */

void (*resumeDevice)(UINT32 hDevice,
                     void * pDriverData);

} USBHST_DEVICE_DRIVER, *pUSBHST_DEVICE_DRIVER;
```
• **Class Specific Devices** - devices that adhere to the standard USB Class Specification supported by [www.usb.org](http://www.usb.org). Class specific devices are identified by class, sub-class and protocol.

• **Vendor Specific Devices** - devices that do not adhere to standard USB Class Specification and their implementation are vendor specific. Vendor specific devices are identified by Vendor ID, Device ID and BCD Protocol.

The Wind River USB host stack 2.0 provides support to register both kinds of devices with the USB Subsystem. The USBD client is notified whenever a device of a particular type is attached to the system or removed from the system. The client can specify callback routines to request this notification.

When a client no longer intends to use the USBD, it must call `usbHstDriverDeregister()` in order to release its per-client data.

---

**Example 3-2  Registration and De-Registration of a USBD Client**

The following code fragment demonstrates the registration and de-registration of a USBD client. The callbacks specified by the client are `DeviceAdd_Callback`, `DeviceRemove_CallBack`, `DeviceSuspend_Callback`, and `DeviceResume_Callback`. These callbacks notify the client of a device attachment, device detachment, device suspend, and a device resume, respectively.

```c
/****************************************************************/
/* Example code for registration and de-registration */
/************************************************************************

pUSBHST_DEVICE_DRIVER pDriverData;
USBHST_STATUS status;

/* allocate structure for driver specific data */
if ( !(pDriverData = OSS_CALLOC (sizeof (USBHST_DEVICE_DRIVER))))
    return ERROR;

/* initialize structure with class, subclass, protocol and callback details */

pDriverData->bFlagVendorSpecific = 0;
pDriverData->uVendorIDorClass = deviceClass;
pDriverData->uProductIDorSubClass = deviceSubClass;
pDriverData->uBCDUSBorProtocol = deviceProtocol;
pDriverData->addDevice = DeviceAdd_Callback;
pDriverData->removeDevice = DeviceRemove_CallBack;
pDriverData->suspendDevice = DeviceSuspend_Callback;
pDriverData->resumeDevice = DeviceResume_Callback;

```
3 USB Host Drivers
3.3 The USB Host Driver (USBD)

pDriverData->resumeDevice = DeviceResume_Callback;

/* register the client with USBD */
status = usbHstDriverRegister (pDriverData, NULL);
if (status != USBHST_SUCCESS)
{
    /* release driver specific structure */
    OSS_FREE (pDriverData);
    return ERROR;
}
/* deregister the client */
status = usbHstDriverDeregister (pDriverData);

Standard Request Interfaces

The USB specification defines a set of standard requests, a subset of which must be supported by all USB devices. Typically, a client uses these routines to interrogate a device’s descriptors—thus determining the device’s capabilities—and to set a particular device configuration.

Some of the standard request interfaces exposed by the USBD are as follows:

- **usbHstGetDescriptor()**
  This routine is used to issue the GET_DESCRIPTOR USB standard request.

- **usbHstSetDescriptor()**
  This routine is used to issue the SET_DESCRIPTOR USB standard request.

- **usbHstGetInterface()**
  This routine is used to issue the GET_INTERFACE USB standard request.

- **usbHstSetInterface()**
  This routine is used to issue the SET_INTERFACE USB standard request.

- **usbHstGetConfiguration()**
  This routine is used to issue the GET_CONFIGURATION USB standard request.

- **usbHstSetConfiguration()**
  This routine is used to issue the SET_CONFIGURATION USB standard request.

- **usbHstClearFeature()**
  This routine is used to issue the CLEAR_FEATURE USB standard request.
**usbHstSetFeature()**
This routine is used to issue the SET_FEATURE USB standard request.

**usbHstGetStatus()**
This routine is used to issue the GET_STATUS USB standard request.

**usbHstSetSynchFrame()**
This routine is used to issue the SYNCH_FRAME USB standard request.

**Example 3-3  Standard Request Interface**

The following example shows the standard request interface **usbHstSetDescriptor()** provided by the USB host stack.

```c
USBHST_STATUS IssueDeviceSetDescriptor
(
    UINT32 hDevice,
    UINT32 uSize,
    UCHAR pBuffer
)
{
    /* To store the setup packet */
    USBHST_SETUP_PACKET SetupPacket;

    /* To store the USB request block */
    USBHST_URB Urb;

    /* To store the usb status returned on submission of request */
    USBHST_STATUS nStatus = USBHST_FAILURE;

    nStatus = usbHstSetDescriptor (hDevice, USBHST_DEVICE_DESC,
                                   0, 0, pBuffer, uSize);

    /* Check the status returned */
    if (USBHST_SUCCESS == nStatus)
    {
        /* Reset the device after set descriptor of device */
    }

    return nStatus;
}
/* END of IssueDeviceSetDescriptor () */
```

**Data Transfer Interfaces**

Once a client has completely configured a device, the client can begin to exchange data with that device using the data transfer interfaces provided by the USBD.
Control, bulk, interrupt, and isochronous transfers are described using a single
USBHST_URB data structure (defined in usbHst.h).

The data transfer interfaces exposed by the USBD are usbHstURBSUBmit() and
usbHstURBCancel().

Class drivers use these interfaces to:

- Submit a data transfer request.
- Cancel a data transfer request.
- Issue class-specific or vendor-specific requests using data transfers on the
default control endpoint.

Example 3-4  Data Transfer Interface

The following example illustrates the data transfer interface usbHstURBSUBmit() provided by USB host stack.

/* Callback routine for blocking calls */
USBHST_STATUS Block_CompletionCallback(PUSBHST_URB pUrb)
{
    /* Check if pContext is valid */
    if ((NULL == pUrb) || (NULL == pUrb->pContext))
    {
        return USBHST_FAILURE;
    }

    /* Release the event(release a semaphore) */
    OS_RELEASE_EVENT((OS_EVENT_ID)pUrb->pContext);
    return USBHST_SUCCESS;
} /* End of routine Block_CompletionCallback */

/* Issue block routine */
USBHST_STATUS IssueBlockBulkSubmitUrb
{
    UINT32 hDevice,
    UINT8 uRequest,
    UINT8 uRequestCode,
    UINT16 uValue,
    UINT16 uIndex,
    UINT32 uSize,
    PUCHAR pBuffer
}

{ /* To store the USB request block */
USBHST_URB Urb;

/* To store the usb status returned on submission of request */
USBHST_STATUS nStatus = USBHST_FAILURE;

/* To store the event id */
OS_EVENT_ID EventId;

/* Allocate memory for urb and reset its values */
..........................
..........................
..........................

/* Create transfer completion event for making a
* blocking call (create a semaphore) */
EventId = OS_CREATE_EVENT(OS_EVENT_NON_SIGNALED);

/* Populate the Urb structure */
USBHST_FILL_BULK_URB(&Urb, hDevice, 0, pBuffer, uSize,
    USBHST_SHORT_TRANSFER_OK, Block_CompletionCallback,
    NULL, USBHST_FAILURE);

/* Call the submitURB routine to submit the URB. */
nStatus = usbHstURBSubmit (&Urb);

/* Check the status */
if (USBHST_SUCCESS == nStatus)
{
    /* Wait for the completion of the event(release of semaphore) */
    OS_WAIT_FOR_EVENT(EventId, OS_WAIT_INFINITE);

    /* Store the status returned by Urb */
    nStatus = Urb.nStatus;
}

/* Update parameters and free the resources */
..........................
..........................

/* Destroy the event(destroy the semaphore) */
OS_DESTROY_EVENT(EventId);

/* Return the status of Urb */
return nStatus;
}

/* END of IssueBlockBulkSubmitUrb () */
3.3.2 USBD 1.1 Compatibility

The Wind River USB host stack provides backward compatibility for class drivers developed using the USB Developer’s Toolkit, 1.1.x product. This backward compatibility is achieved using a thin software layer called a translation unit. The translation unit provides the USBD interface from the USB Developer’s Toolkit, 1.1.x on top of the USB host stack, 2.1. In this document, the USBD interface provided by USB Developer’s Toolkit, 1.1.x is referred to as the USBD 1.1 interface.

Initializing the USBD

Using the USBD 1.1 interface, initializing the USBD is a two-step process. First, the USBD entry point, usbdInitialize() is called at least once. The usbdInitialize() routine initializes internal USBD data structures and, in turn, calls the initialization entry points of other modules in the USB host stack such as the hub class driver. In a given system, it is acceptable to call usbdInitialize() once (during the boot sequence for example) or to call it many times (such as during the initialization of each USBD client).

The following code fragment illustrates the proper initialization and shutdown of the USBD:

```c
...  /* Initialize the USBD. */
  if (usbdInitialize () != OK)
    return ERROR;
...  ...
  /* application code. */
  ...
...  /* Shut down the USBD. Note: we only execute this code if the return
    * code from usbdInitialize() was OK. Note also that there is really no
    * reason to check the result of usbdShutdown().
    */
  usbdShutdown ();
```

The second step of the USBD initialization is to attach at least one HCD to the USBD. Normally, the attachment of HCDs to the USBD happens during the VxWorks boot sequence. However, the USBD is designed to allow HCDs to be attached (and removed) at run-time. This flexibility allows the USBD to support systems in which USB host controllers can be “hotswapped” without restarting the system.
Order of Initialization

The `usbdInitialize()` routine must be called before all other USBD routines. However, it is not necessary for all USBD clients to call `usbdInitialize()` before one or more HCDs are attached to the USBD. Both of the following initialization scenarios are acceptable:

**Scenario #1: Traditional Boot Time Initialization**

The order of initialization is as follows:

1. Call `usbdPciInit()`.
2. Call `usbdInit()`.
3. Call `usbHubInit()`.
4. Call `usbOhciInit()`, `usbEhcdInit()`, or `usbUhcdInit()` to attach to the respective HCD.
5. Call the USB class driver initialization entry point.
6. The USB class driver calls `usbdInitialize()`.

**Scenario #2: Hotswap-Driven Initialization**

1. Call `usbdPciInit()`.
2. Call the USB class driver initialization entry point.
3. The USB class driver calls `usbdInit()` and `usbHubInit()`.
4. Hotswap code identifies the presence or insertion of the USB host controller.
5. Hotswap code calls `usbOhciInit()`, `usbEhcdInit()`, or `usbUhcdInit()` to attach to the respective HCD.
6. The USB class driver calls `usbdInitialize()`.

Similarly, calls to `usbOhciExit()`, `usbEhcdExit()`, and `usbUhcdExit()` can be made at any time to detach the respective HCDs from the USBD.

⚠️ **CAUTION:** While the USBD and Wind River HCDs are designed to support the dynamic attachment and detachment of HCDs, certain Wind River BSPs may not support the disconnection of the USB HCD from the underlying PCI interrupt vector, which is essential to the detaching process. If the vector is subsequently re-enabled, the system may attempt to execute a stale interrupt service routine and will fault.
To understand why these different scenarios work correctly, it is necessary to recognize that hot-plugging of USB devices can occur at any time. So, both the USBD and its clients must be written (using the dynamic attachment routines described in *Dynamic Attachment Registration*, p.49) to recognize the appearance and disappearance of USB devices at run-time. It is for this reason that the appearance and disappearance of host controllers can also be handled flexibly. When a new host controller is attached to the system, the USBD automatically identifies the devices connected to the controller and notifies the necessary clients.

Likewise, when a host controller is removed from the system, the USBD automatically notifies the necessary clients that the devices attached to that host controller have been disconnected. The key point here is that USBD clients, such as USB class drivers, never assume that a particular device is present when the client is first initialized, and these drivers are always ready to accept devices connected to the system at other times.

**Registering Client Modules**

Client modules that intend to make use of the USBD to communicate with USB devices must, in addition to calling `usbdInitialize()` , register with the USBD by calling `usbdClientRegister()` . When a client registers with the USBD, the USBD allocates per-client data structures that are later used to track all requests made by that client. During client registration, the USBD also creates a callback task for each registered client (see *Client Callback Tasks*, p.48). After successfully registering a new client, the USBD returns a handle, `USBD_CLIENT_HANDLE`, that must be used by that client when making subsequent calls to the USBD.

When a client no longer intends to use the USBD, it must call `usbdClientUnregister()` in order to release its per-client data and callback task. Any outstanding USB requests made by the client are canceled at that time.

**Example 3-5  Registration and De-Registration of the USBD Client**

The following code fragment demonstrates the registration and de-registration of a USBD client:

```c
USBD_CLIENT_HANDLE usbdClientHandle;

/* Register a client named "USBD_TEST" with the USBD. */
if (usbdClientRegister("USBD_TEST", &usbdClientHandle) != OK)
    { /* Attempt to register a new client failed. */
        return ERROR;
    }
```

---

47
/* Client is registered...application code follows. */
...
/* Unregister the client. */
usbdClientUnregister (usbdClientHandle);

Client Callback Tasks

USB operations can be time-critical. For example, both USB interrupt and isochronous transfers depend on timely servicing in order to work correctly. In a host system in which several different USBD clients are present, the possibility always exists for one client to interfere with the timely execution of other clients needing to service time-sensitive USB traffic. The Wind River USBD introduces per-client callback tasks to manage this problem.

Many USB events can result in callbacks to a USBD client. For example, whenever the USB completes the execution of a USB I/O request packet (IRP), the client’s IRP callback routine is invoked. Similarly, whenever the USB recognizes a dynamic attachment event, one or more client’s dynamic attachment callback routines are invoked. Instead of invoking these callback routines immediately, the USBD schedules the callbacks to be performed by the callback task for the appropriate USBD client(s). Normally, the callback task for each client is “dormant” (in a blocked state). When the USBD schedules a callback for a specified client, the corresponding client callback task “wakes up” (unblocks) and performs the actual callback. This approach allows the USBD to process all outstanding USB events before the clients themselves obtain control of the CPU.

The callback task for each client inherits the VxWorks task priority of the task that originally called usbdClientRegister(). This ensures that callbacks are processed at the task priority level intended by each client and allows you to write clients to take advantage of task priorities as a means of ensuring proper scheduling of time-sensitive USB traffic.

Because each client has its own callback task, clients have greater flexibility in the amount of work they can do during the callback. For example, during callback, it is acceptable for executing code to block without hurting the performance of the USBD or other USBD clients.

Client callback tasks have the VxWorks task name tUsbdCln.
Dynamic Attachment Registration

A typical USBD client wants to be notified whenever a device of a particular type is attached to or removed from the system. By calling the `usbdDynamicAttachRegister()` routine, a client can specify a callback routine to request such notification.

USB device types are identified by a class, subclass, and protocol (in case of class specific devices) and by vendor ID, product ID and bcdDevice (in case of vendor specific devices). Standard USB classes are defined in `usb.h` as `USB_CLASS_XXXX`. Subclass and protocol definitions depend on the class; therefore, these constants are generally defined in the header files associated with a specific class. Sometimes, a client is interested in a narrow range of devices. In this case, it specifies values for the class, subclass, and protocol (for class specific devices) and specifies vendor ID, product ID and bcdDevice (for vendor specific devices) when registering through `usbdDynamicAttachRegister()`.

For example, the USB keyboard class driver, `usbKeyboardLib`, registers for a human interface device (HID) class of `USB_CLASS_HID`, a subclass of `USB_SUBCLASS_HID_BOOT`, and a protocol of `USB_PROTOCOL_HID_BOOT_KEYBOARD` (both the subclass and protocol are defined in `usbHid.h`). In response, by means of the callback mechanism, the USBD notifies the keyboard class driver whenever a device matching exactly this criterion is attached to or removed from the system.

In other cases, a client’s interest is broader. In this case, the constant `USBD_NOTIFY_ALL` (defined in `usbdLib.h`) can be substituted for any or all of the class, subclass, and protocol match criteria. For example, the USB printer class driver, `usbPrinterLib`, registers for a class of `USB_CLASS_PRINTER`, subclass of `USB_SUBCLASS_PRINTER` (defined in `usbPrinter.h`), and a protocol of `USBD_NOTIFY_ALL`.

While a typical client makes only a single call to `usbdDynamicAttachRegister()` , there is no limit to the number of concurrent notification requests a client can have. A single client can register concurrently for attachment notification of as many device types as desired.

Example 3-6 Dynamic Attachment Registration Routines

The following code fragments demonstrates the correct use of the dynamic attachment registration routines:

```c
/****************************************************************************
* attachCallback - called by USBD when a device is attached/removed
*/
```
* The USBD invokes this callback when a USB device is attached to or
* removed from the system. <nodeId> is the USBD_NODE_ID of the node being
* attached or removed. <attachAction> is USBD_DYNA_ATTACH or
* USBD_DYNA_REMOVE.
* RETURNS: N/A
*/
LOCAL VOID attachCallback
(
USBD_NODE_ID nodeId,
UINT16 attachAction,
UINT16 configuration,
UINT16 interface,
UINT16 deviceClass,
UINT16 deviceSubClass,
UINT16 deviceProtocol)
{
    /* Depending on the attachment code, add or remove a device. */
    switch (attachAction)
    {
    case USBD_DYNA_ATTACH:
        /* A device is being attached. */
        printf("New device attached to system.\n");
        break;
    case USBD_DYNA_REMOVE:
        /* A device is being detached. */
        printf("Device removed from system.\n");
        break;
    }
}

During the initialization of the application, the following code fragment also appears:

/*
 * Register for dynamic notification when a USB device is
 * attached to or removed from the system.
 * For the sake of demonstration, we'll request notification for
 * USB printers, though this same code could be used for
 * any other type of device.
 *
 * usbdClientHandle is the USBD_CLIENT_HANDLE for the client.
 * USB_CLASS_PRINTER is defined in usb.h. USB_SUBCLASS_PRINTER
 * is defined in usbPrinter.h. USB_NOTIFY_ALL is a wild-card
 * that matches anything. In this case we use it to match any
 * USB programming interface.
 */
if (usbdDynamicAttachRegister (usbdClientHandle, USB_CLASS_PRINTER,
    USB_SUBCLASS_PRINTER, USB_NOTIFY_ALL, FALSE, attachCallback) != OK)
{
    /* Attempt to register for dynamic attachment notification failed.*/
    return ERROR;
}

/*
 * attachCallback() - above - is now called whenever a USB printer
3.3 The USB Host Driver (USBD)

The following is the API definition for `usbdDynamicAttachRegister()`:

```c
STATUS usbdDynamicAttachRegister
    (USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    UINT16 deviceClass, /* USB class code */
    UINT16 deviceSubClass, /* USB sub-class code */
    UINT16 deviceProtocol, /* USB device protocol code */
    BOOL vendorSpecific, /* for vendor specific devices */
        * TRUE for vendor specific devices
        * FALSE for class specific devices */
    USBD_ATTACH_CALLBACK attachCallback /* User-supplied callback */
    )
```

A client can specify if it wants to receive notification for only vendor specific devices. In such a case, the 'vendorSpecific' flag during registration is set to TRUE. The `deviceClass`, `deviceSubClass` and `deviceProtocol` are set to `vendorID`, `deviceID` and `bcdDevice`, respectively.

In case the client wants to receive notification for class specific devices, the 'vendorSpecific' flag is set to FALSE during the call to `usbdDynamicAttachRegister()`. The `deviceClass`, `deviceSubClass` and `deviceProtocol` are set to `class`, `subclass` and `protocol`, respectively.

The following code fragment demonstrates the registration of the callback using `usbdDynamicAttachRegister()` for class specific and vendor specific devices.

```c
/* for class specific devices */
usbdDynamicAttachRegister (usbdClientHandle,
    class,
    subclass,
    protocol,
    FALSE,
    callbackFunction
    );

/* for vendor specific devices */
```
usbdDynamicAttachRegister (usbdClientHandle,
  vendorID,
  productID,
  bcdDevice,
  TRUE,
  callbackFunction
 )

Node IDs

USB devices are always identified using a **USBD_NODE_ID**. The **USBD_NODE_ID** is, in effect, a handle created by the translation unit to track a device (the node ID was created by the USBD in USB 1.1); it has no relationship to the device’s actual USB address. This reflects the fact that clients are usually not interested in knowing to which USB/host controller a device is physically attached. Because each device is referred to using the abstract concept of a node ID, the client can remain unconcerned with the details of physical device attachment and USB address assignment, and the USBD can manage these details internally.

When a client is notified of the attachment or removal of a device, the USBD always identifies the device in question using **USBD_NODE_ID**. Likewise, when the client wishes to communicate through the USBD with a particular device, it must pass **USBD_NODE_ID** for that device to the USBD.

Bus Enumeration Routines

The routines **usbdBusCountGet()**, **usbdRootNodeIdGet()**, **usbdHubPortCountGet()**, **usbdNodeIdGet()**, and **usbdNodeInfoGet()**—formerly provided by the **usbdLib** module in USB 1.1—are provided in USB 2.0 for backward compatibility with the USB 1.1 host stack. As a group, these are referred to as bus enumeration routines, and they allow USBD clients to enumerate the network of devices attached to each host controller.

These routines are useful in the authoring of diagnostic tools and test programs such as **usbTool**. However, when the caller uses these routines, it has no way of knowing if the USB topology changes after enumeration or even mid-enumeration. Therefore, authors of traditional clients, such as USB class drivers, are advised not to use these routines.

Device Configuration

The USB specification defines a set of standard requests, a subset of which must be supported by all USB devices. Typically, a client uses the following routines to
interrogate a device’s “descriptors”—thus determining the device’s capabilities—and to set a particular device configuration:

The USBD 1.1 standard routines are as follows:

- **usbdFeatureClear()** - clears a USB Feature
- **usbdFeatureSet()** - sets a USB Feature
- **usbdConfigurationGet()** - gets the current configuration from the device
- **usbdConfigurationSet()** - sets the configuration of the device
- **usbdDescriptorGet()** - gets the device, configuration, interface, endpoint and string descriptors from the device
- **usbdDescriptorSet()** - sets the device, configuration, interface, endpoint and string descriptors on the device
- **usbdInterfaceGet()** - gets the current alternate setting for the given device interface.
- **usbdInterfaceSet()** - sets the alternate setting for the given device interface.
- **usbdStatusGet()** - retrieves the status from a USB device, interface or endpoint
- **usbdSynchFrameGet()** - retrieves the device frame numbers.

**NOTE:** For more details on these APIs, refer to *USB API Reference.*

The USBD itself takes care of certain USB configuration issues automatically. Most critically, the USBD internally implements the code necessary to manage USB hubs and to set device addresses when new devices are added to the USB topology. Clients must not attempt to manage these routines themselves, for doing so is likely to cause the USBD to malfunction.

The USBD also monitors configuration events. When a USBD client invokes a USBD routine that causes a configuration event, the USBD automatically resets the USB data toggles (DATA0 and DATA1) associated with the device’s pipes and endpoints. For example, a call to the USB command **usbdConfigurationSet()** issues the set configuration command to the device, and in the process resets the data toggle to DATA0. Because the USBD handles USB data toggles automatically, you do not typically need to concern yourself with resetting data toggles. For an explanation of pipes and endpoints in the USB environment, see *Data Flow,* p.56. For more information about USB data toggles, see the USB specification.
Example 3-7 **Reading and Parsing a Device Descriptor**

Device configuration depends heavily on the type of device being configured. The following code fragment demonstrates how to read and parse a typical device descriptor:

```c
/*******************************************************************/
/* USB_MAX_DESCR_LEN defined in usb.h */
UINT8 bfr [USB_MAX_DESCR_LEN];

UINT16 actLen;

/* USB_CONFIG_DESCR defined in usb.h */
pUSB_CONFIG_DESCR pCfgDescr;

/* USB_INTERFACE_DESCR is also in usb.h */
pUSB_INTERFACE_DESCR pIfDescr;

/* Read the configuration descriptor. In the following fragment
 * it is assumed that nodeId was initialized, probably in response
 * to an earlier call to the client’s dynamic attachment
 * notification callback
 * (see above).
 */

if (usbdDescriptorGet (usbdClientHandle, nodeId,
USB_RT_STANDARD | USB_RT_DEVICE, USB_DESCR_CONFIGURATION, 0, 0,
sizeof (bfr), bfr, &actLen) != OK)
{
    /* We failed to read the device’s configuration descriptor. */
    return FALSE;
}

/* Use the routine usbDescrParse() - exported by usbLib.h -
 * to extract the configuration descriptor and the first
 * interface descriptor from the buffer.
 */

if ((pCfgDescr = usbDescrParse (bfr, actLen, USB_DESCR_CONFIGURATION))
== NULL)
{
    /* No configuration descriptor was found in the buffer. */
    return FALSE;
}

if ((pIfDescr = usbDescrParse (bfr, actLen, USB_DESCR_INTERFACE))
== NULL)
{
    /* No interface descriptor was found in the buffer. */
    return FALSE;
}
```
Pipe Creation and Deletion

USB data transfers are addressed to specific endpoints within each device. The channel between a USBD client and a specific device endpoint is called a pipe. Each pipe has a number of characteristics, including:

- the USBD_NODE_ID of the device
- the endpoint number on the device
- the direction of data transfer
- transfer type
- maximum packet size
- bandwidth requirements
- latency requirements

In order to exchange data with a device, a client must first create a pipe. In response to a client request to create a new pipe, the USBD creates a USBD_PIPE_HANDLE, which the client must use for all subsequent operations on the pipe.

Example 3-8 Creating a Pipe for Sending Data on a Printer

The following code fragment demonstrates the creation of a pipe for sending data to the bulk output endpoint on a printer:

```c
USBD_PIPE_HANDLE outPipeHandle;
/* Create a pipe for output to the printer.
 It is assumed that endpoint, configValue, interface,
 * and maxPacketSize were determined by reading
 * the appropriate descriptors from the device.
 */
if (usbdPipeCreate (usbdClientHandle, nodeId, endpoint, configValue,
 interface, USB_XFRTYPE_BULK, USB_DIR_OUT, maxPacketSize, 0, 0,
 &outPipeHandle) != OK)
{
 /* We failed to create the pipe. */
 return ERROR;
}
```

If the pipe is no longer required, it must be destroyed. Destroying a pipe removes all references for that pipe, and no further data transfer can occur on that pipe.

1. Unlike IEEE-1394, the USB treats an isochronous transfer as an exchange between the USB host and a specific USB device. In contrast, 1394 treats isochronous transfers as a broadcast to which any number of 1394 devices can listen simultaneously.
order to destroy the pipe, one must specify the `USBD_CLIENT_HANDLE` (handle to the client module) and `USBD_PIPE_HANDLE` (handle to the pipe).

Example 3-9  Deleting a Pipe

The following code fragment demonstrates the deletion of pipe for a bulk out endpoint on a printer:

```c
/***********************************************************/
USBD_PIPE_HANDLE outPipeHandle;
USBD_CLIENT_HANDLE usbdClientHandle;

/* Destroys the pipe */
if (usbdPipeDestroy (usbdClientHandle, outPipeHandle) != OK)
{
    /* We failed to destroy the pipe. */
    return ERROR;
}
```

**NOTE:** When the `usbdPipeDestroy()` routine is called, it first looks for any IRP that is pending on the endpoint to be transferred. If so, it first aborts all the pending transfers on the endpoint, and then destroys the pipe.

Data Flow

Once a client has completely configured a pipe, it can begin to exchange data with that device using the pipe and transfer routines provided by the USBD.
3 USB Host Drivers
3.3 The USB Host Driver (USBD)

Figure 3-2  State-Level Diagram of Data Flow

1. Device Driver creates IRPS
2. USBHST_HC_DRIVER submits URB (pUrb)
3. Copy data to buffer
4. Create HC-specific data structure
5. OUT
6. DATA
7. ACK
8. Interrupt generated
9. Sets URB:: result = USBHST_SUCCESS
10. usbtuDataUrbCompleteCallback(pUbb)
11. Calls the client Callback
12. Sets IRB:: result = SUCCESS
13. usbtuDataUrbCompleteCallback(pUbb)
14. Creates URBs and registers
15. usbtuDataUrbCompleteCallback as callback
16. usbTransfer (clientHandle, pipeHandle, &irps)
17. Device Driver translation unit USBD creates URBs and registers
18. usbtuDataUrbCompleteCallback as callback
19. USBHST_HC_DRIVER submits URB (pUrb)
20. Copy data to buffer
21. Create HC-specific data structure
22. OUT
23. DATA
24. ACK
25. Interrupt generated
26. Sets URB:: result = USBHST_SUCCESS
27. usbtuDataUrbCompleteCallback(pUbb)
28. Calls the client Callback
29. Sets IRB:: result = SUCCESS
30. usbtuDataUrbCompleteCallback(pUbb)
31. Device Driver creates IRPS
Control, bulk, interrupt, and isochronous transfers are described using a single USB_IRP data structure (defined in usb.h). Figure 3-2 illustrates the data flow.

The usbdTransfer() interface is used to transfer the data between the host and the device.

Example 3-10 Data Transfer on a Pipe

The following code fragment demonstrates the data transfer on a pipe for a bulk out endpoint on a printer:

```
/*******************************************************************/
UINT8 bfr [4096]; /* a buffer of arbitrary size. */
UINT16 bfrCount; /* amount of data in the buffer. */
USB_IRP irp;

/* The code here would presumably put data into the buffer 
 * and initialize bfrCount with the amount of data in the buffer. 
 */
... 
/* Initialize IRP. */
memset (&irp, 0, sizeof (irp));
irp.userPtr = ourUserPtr; /* a value to use to our callback */
irp.irpLen = sizeof (IrP); /* the length of the IRP */
irp.userCallback = ourIrpCallback; /* our IRP completion callback */
irp.timeout = 30000; /* 30 second timeout */
irp.transferLen = bfrCount;
irp.bfrCount = 1;
irp.bfrList [0].pid = USB_PID_OUT;
irp.bfrList [0].pBfr = bfr;
irp.bfrList [0].bfrLen = count;

/* Submit IRP */
if (usbdTransfer (usbdClientHandle, outPipeHandle, &irp) != OK)
{
    /* An error here means that our IRP was malformed. */
    return ERROR;
}
```

The usbdTransfer() function returns as soon as the IRP has been successfully enqueued. If there is a failure in delivering the IRP to the HCD, then usbdTransfer() returns an error. The actual result of the IRP should be checked after the userCallback routine has been invoked.

The class driver may submit the next IRP, depending upon the result of the callback routine. It may resubmit the IRP or call usbdTransferAbort() routine to cancel all submitted IRPs.
3.4 Host Controller Drivers

This section describes the interface and requirements for host controller drivers (HCDs). The Wind River USB host stack comes with EHCI, OHCI, and UHCI drivers. This section is essential if you are creating an HCD for a host controller that Wind River does not already support. This section also explains USBD data structures and USBD interfaces used by the host controller driver.

3.4.1 Registering the Host Controller Driver

All HCDs are required to register with the USBD by calling the routine `usbHstHCDRegister()`. This routine takes the following form:

```c
USBHST_STATUS usbHstHCDRegister
(
    pUSBHST_HC_DRIVER pHCDriver,
    UINT32 *phHCDriver,
    void * pContext
);
```

3.4.2 USHST_HC_DRIVER Structure

The `USHST_HC_DRIVER` structure contains the host controller driver function pointers. During host controller driver initialization, this structure is populated and registered with the USB host stack. The USBD uses the function pointers in this structure to communicate with the host controller driver.

**Example 3-11 USHST_HC_DRIVER Structure**

This structure takes the following form:

```c
/*******************
/* This structure contains the HC driver function pointers */
typedef struct usbhst_hc_driver
{
    /* Number of bus for this host controller */
    UINT8 uNumberOfBus;

    /* Function pointer to get the frame number */
    USBHST_STATUS (*getFrameNumber)(UINT8 uBusIndex,
                                      UINT16 *puFrameNumber);

    /* Function pointer to set the bit rate */
};
```
USBHST_STATUS (*setBitRate) (UINT8 uBusIndex,
                       BOOL bIncrement,
                       UINT32 *puCurrentFrameWidth);
/* Function pointer to check if required bandwidth is available */

USBHST_STATUS (*isBandwidthAvailable) (UINT8 uBusIndex,
                                      UINT8 uDeviceAddress,
                                      UINT8 uDeviceSpeed,
                                      UCHAR *pCurrentDescriptor,
                                      UCHAR *pNewDescriptor);
/* Function pointer to create a pipe */

USBHST_STATUS (*createPipe) (UINT8 uBusIndex,
                            UINT8 uDeviceAddress,
                            UINT8 uDeviceSpeed,
                            UCHAR *pEndPointDescriptor,
                            UINT16 uHighSpeedHubInfo,
                            UINT32 *puPipeHandle);
/* Function pointer to modify the default pipe */

USBHST_STATUS (*modifyDefaultPipe) (UINT8 uBusIndex,
                                   UINT32 uDefaultPipeHandle,
                                   UINT8 uDeviceSpeed,
                                   UINT8 uMaxPacketSize,
                                   UINT16 uHighSpeedHubInfo,
                                   UINT32 *puPipeHandle);
/* Function pointer to delete a pipe */

USBHST_STATUS (*deletePipe) (UINT8 uBusIndex,
                           UINT32 uPipeHandle);
/* Function pointer to check if there are any
requests pending on the pipe */

USBHST_STATUS (*isRequestPending) (UINT8 uBusIndex,
                                    UINT32 uPipeHandle);
/* Function pointer to submit a USB request */

USBHST_STATUS (*submitURB) (UINT8 uBusIndex,
                         UINT32 uPipeHandle,
                         pUSBHST_URB pURB);
/* Function pointer to cancel USB request */

USBHST_STATUS (*cancelURB) (UINT8 uBusIndex,
                        UINT32 uPipeHandle,
                        pUSBHST_URB pURB);
/* Function pointer to submit the status of the clear TT request */

USBHST_STATUS (*clearTTRequestComplete)
3.4.3 Host Controller Driver Interfaces

All requests to the HCD are made by the USBD which calls function pointers registered with the USBD. This subsection explains each of these routines.

USBHST_HC_DRIVER Structure

The following routine prototypes are in the structure USBHST_HC_DRIVER.

modifyDefaultPipe()

This routine modifies the properties of the default pipe (address 0, endpoint 0).

isBandwidthAvailable()

This routine determines if there is enough bandwidth to support the new configuration or an alternate interface setting. The routine parses through the configuration or interface descriptor. When the configuration descriptor is passed as a parameter, it includes all interface and endpoint descriptors. Likewise, when the interface descriptor is passed as a parameter, it includes all of the endpoint descriptors corresponding to that interface.

createPipe()

This routine parses and creates a pipe for the specified USB endpoint descriptor. Upon creation, the pipe is added to a queue of pipes maintained for transfers.

deletePipe()

This routine deletes a pipe corresponding to an endpoint. This routine searches the list of pipes for the pipe corresponding to a given endpoint. The selected pipe is then deleted from the list and the list is updated. All pending transfers for the pipe are cancelled and the callback routines for the transfers are called.

submitURB()

This routine submits a request to the endpoint. The routine populates any host-controller-specific data structures to submit a transfer. The new request is
added to the queue of requests pending for the endpoint. This routine is implemented as a non-blocking routine. When a request completes, the callback routine for the request is called.

cancelURB()
This routine cancels a request submitted to an endpoint. The routine searches the queue of requests pending for a given endpoint. If the desired request is pending for the endpoint, the request is cancelled and the queue of requests for the endpoint is updated.

isRequestPending()
This routine checks if there are any requests pending for an endpoint. The routine looks for any requests pending for the endpoint and returns accordingly. If there are pending requests, a success status is returned.

getFrameNumber()
This routine obtains the current frame number. The routine reads the frame number register and returns the same.

setBitRate()
This routine modifies the frame width. The frame width can be incremented or decremented by one bit time once in every six frames. If the functionality is supported, this routine modifies the frame width.

clearTTRequestComplete() and resetTTRequestComplete()
These routines are used to handle errors that occur during split transaction. These routines are implemented only for an EHCI host controller. The EHCD registers these functions with the USBD during initialization.

**USBHST_USBD_TO_HCD_FUNCTION_LIST Structure**

The EHCD maintains one more interface called **USBHST_USBD_TO_HCD_FUNCTION_LIST** defined in the file /target/h/usb2/usbHst.h.

```c
typedef struct usbHstUsbdToHcdFunctionList {
    /*
     * Pointer to function, which will be called
     * by the Host Controller Driver to submit clear TT request
     */
    USBHST_STATUS (*clearTTRequest)(UINT32 hHCDriver,
                                     UINT8 uRelBusIndex,
                                     UINT8 uHighSpeedHubAddress,
                                     UINT8 uHighSpeedPortNumber,
                                     UINT16 wValue,
                                     void * pContext);
```

62
3 USB Host Drivers

3.4 Host Controller Drivers

```c
/*
 * Pointer to function, which will be called by
 * Host Controller Driver to submit reset TT request
 */
USBHST_STATUS (*resetTTRequest)(UINT32 hHCDriver,
   UINT8 uRelBusIndex,
   UINT8 uHighSpeedHubAddress,
   UINT8 uHighSpeedPortNumber,
   void * pContext);

} USBHST_USBD_TO_HCD_FUNCTION_LIST,
*pUSBHST_USBD_TO_HCD_FUNCTION_LIST;
```

This data structure is registered with HCD by the USBD.

/* Register the HCD with the USBD */
Status = usbHstHCDRegister(g_pEHCDriverInfo, &g_EHCDHandle,
   &g_USBDRequestFunctions);

If an error occurs on the hub, the EHCD will not handle any request for that hub unless the error is handled, that is, the TT buffers are cleared.

Figure 3-3 is a state diagram that explains how the split errors are handled using above interface.

NOTE: The Reset_TT request is handled similarly.

3.4.4 Registering a Bus for the Host Controller

All HCDs are required to register a bus for the host controller with the USB host stack by calling the routine usbHstBusRegister(). For example, assuming that there are three OHCI host controllers on the system:

1. The usbHstHCDRegister() routine is called once to register the OHCI host controller driver.

2. The usbHstBusRegister() routine is called three times to register the three OHCI host controllers.
3.4.5 De-registering the Bus for the Host Controller

All HCDs de-register the bus for the host controller with the USB host stack by calling the `usbHstBusDeregister()` routine. This routine is called for the number of the bus registered. If there are functional devices (active devices or configured devices) on the bus, the bus is not de-registered.

3.4.6 De-registering the Host Controller Driver

All HCDs de-register the host controller driver from the USB host stack by calling the `usbHstHCDDeregister()` routine. If there are buses registered for the host controller driver, the host controller driver is not de-registered.
3.4.7 HCD Error Reporting Conventions

You are encouraged to use the existing HCD error codes when creating new HCDs. According to Wind River convention, the HCD sets the system `errno` whenever returning an error. HCDs return the standard VxWorks constant `USBHST_SUCCESS` when a routine completes successfully. HCD error codes are defined as follows:

```
USBHST_MEMORY_NOT_ALLOCATED
USBHST_INSUFFICIENT_BANDWIDTH
USBHST_INSUFFICIENT_RESOURCE
USBHST_INVALID_REQUEST
USBHST_INVALID_PARAMETER
USBHSTSTALL_ERROR
USBHST_DEVICE_NOT_RESPONDING_ERROR
USBHST_DATA_OVERRUN_ERROR
USBHST_DATA_UNDERRUN_ERROR
USBHST_BUFFER_OVERRUN_ERROR
USBHST_BUFFER_UNDERRUN_ERROR
USBHST_TRANSFER_CANCELLED
USBHST_TIMEOUT
USBHST_BAD_START_OF_FRAME
```

3.4.8 Root Hub Emulation

HCDs are required to emulate the behavior of the root hub. That is, HCDs must intercept transfer requests intended for the root hub and synthesize standard USB responses to these requests. For example, when a host controller is first initialized, the root hub must respond at the default USB address 0, and its downstream ports must be disabled. The USBD interrogates the root hub, just as it would other hubs, by issuing USB GET_DESCRIPTOR requests. If the hub class driver is registered, the USBD calls the hub class driver to configure the root hub by issuing a series of SET_ADDRESS, SET_CONFIGURATION, SET_FEATURE, and CLEAR_FEATURE
requests. The HCD recognizes which of these requests are intended for the root hub and responds to them appropriately.

After configuration, the hub class driver begins polling the root hub’s interrupt status pipe to monitor changes on the root hub’s downstream ports. The HCD intercepts IRPs directed to the root hub’s interrupt endpoint and synthesizes the appropriate replies. Typically, the HCD queues requests directed to the root hub separately from those that actually result in bus operations.
4 USB Class Drivers

4.1 Introduction  67
4.2 Hub Class Driver  68
4.3 Keyboard Driver  72
4.4 Mouse Driver 80
4.5 Printer Driver 84
4.6 Speaker Driver 88
4.7 Mass Storage Class Driver  93
4.8 Communication Class Drivers  99

4.1 Introduction

This chapter provides information on the USB class drivers that are provided with the Wind River USB host stack. Pre-built class drivers for several USB device types are included in your installation.
4.2 Hub Class Driver

Hubs are a specific class of USB devices that provide extension ports for connecting more USB devices to the USB host. The USB hub class driver controls the functionality of a USB hub, including the root hub which is emulated by the host controller driver. The hub class driver provides two interfaces to the USB class drivers and USB applications. The first, `usbHubInit()`, is used to initialize the hub class driver. The second, `usbHubExit()`, is used to exit the hub class driver. The hub class driver uses the USBD APIs to interact with the USB host stack.

4.2.1 Registering the Hub Class Driver

Like other class drivers, the hub class driver registers with the USBD by calling `usbHstDriverRegister()`. By registering with the USBD, the hub class driver provides an interface to the USBD for adding, removing, suspending, and resuming a device. Figure 4-1 illustrates the registration process.

![Figure 4-1 Registering the Hub Class Driver](image-url)
After registering the hub class driver with the USBD, the driver configures the root hub for all of the host controller drivers registered with the USBD. Then, the hub class driver starts polling the root hub’s interrupt pipe to monitor changes on the downstream ports.

4.2.2 Connecting a Device to a Hub

When a device is connected to a hub, the hub class driver is informed by the USB host stack. The interaction shown in Figure 4-2 occurs between the hub class driver, the USB host stack, the hub, and the device.
Figure 4-2  Connecting a Device to a Hub

- Status change interrupt IN Callback is called
- Gets the port status
- Clears the port status
- Resets the port
- Submit the next interrupt IN request
- Status change interrupt IN Callback is called
- Signals of a new device detected
- Returns the device handle
- Configure device
- Checks for PowerLoad of the device
- Configure Device returns
- Submit the next interrupt IN request
4.2.3 Removing a Device From a Hub

When a device is disconnected from a hub, the interaction shown in Figure 4-3 occurs between the hub class driver, the USB host stack, the hub, and the device.

Figure 4-3 Removing a Device from a Hub

4.2.4 De-registering the Hub Class Driver

The hub class driver de-registers from the USB host stack by calling the `usbHstDriverDeregister()` routine. After calling the routine, the hub class driver stops polling the root hub’s interrupt pipe for changes on the downstream port. Figure 4-4 illustrates the de-registration process.
4.3 Keyboard Driver

USB keyboards are described in human interface device (HID)-related specifications. The Wind River implementation of the USB keyboard driver, `usbKeyboardLib`, is concerned only with USB devices claiming to be keyboards as set forth in the USB specification; the driver ignores other types of HIDs.

**NOTE:** USB keyboards can operate according to either a boot protocol or a report protocol. However, the `usbKeyboardLib` driver enables keyboards for operation using the boot protocol only.

4.3.1 SIO Driver Model

The USB keyboard class driver, `usbKeyboardLib`, follows the VxWorks serial I/O (SIO) driver model, with certain exceptions and extensions. As the SIO driver model presents a fairly limited, byte-stream-oriented view of a serial device, the keyboard driver maps USB keyboard scan codes into appropriate ASCII codes.
Scan codes and combinations of scan codes that do not map to the ASCII character set are suppressed.

### 4.3.2 Initializing the Keyboard Class Driver

`usbKeyboardLib` must be initialized by calling `usbKeyboardDevInit()`, which in turn initializes the internal resources of the keyboard class driver and registers the keyboard class driver with the lower layers.

However, before a call to `usbKeyboardDevInit()`, the caller must ensure that the USBD has been properly initialized. Please refer to 2.5 Initializing the USB Host Stack Hardware, p.23 for how to initialize USB Host stack.

The different client applications can call `usbKeyboardDevInit()` multiple times, but the driver internally maintains a usage counter. All the initialization of data structures and registering with the lower layer happens on the very first call to the `usbKeyboardDevInit()`. In this call, the usage counter is incremented from 0 to 1. For all subsequent calls to the initialization routine, the usage counter is simply incremented.

The following code example demonstrates how this is accomplished:

```c
/*
 * initializing the keyboard class driver
 */

STATUS usbKeyboardDevInit (void)
{
    /* If not already initialized, then initialize
    * internal structures and connection to USBD.
    */
    if (initCount == 0)
    {
        /*
         * initialize the data structures and register with
         * the lower layer
         */
        
        /* increment the usage count */
        initCount++;
        return OK;
    }
}
4.3.3 Registering the Keyboard Class Driver

The keyboard class driver registers with the USBD by calling `usbdClientRegister()`. In response to this call, the Translation Unit allocates keyboard-client data structures and a keyboard client callback task.

Once the keyboard class driver is successfully registered with the translation unit, the translation unit returns a handle of type `USBD_CLIENT_HANDLE` which is used for all subsequent communication with the lower layers.

After the keyboard class driver is successfully registered, the driver registers a callback routine with the lower layer by calling the routine `usbdDynamicAttachRegister()`. The callback is called whenever a device of particular class, subclass and protocol is connected to or disconnected from the USB Subsystem. The callback routine takes the following form:

```c
typedef VOID (*USBD_ATTACH_CALLBACK)(
    USBD_NODE_ID nodeId, /* USBD Handler */
    UINT16 attachAction, /* notification for device 
        * attachment or removal 
        * USBD_DYNA_ATTACH or 
        * USBD_DYNA_REMOVE */
    UINT16 configuration, /* configuration index * /
    UINT16 interface, /* interface number */
    UINT16 deviceClass, /* Class ID */
    UINT16 deviceSubClass, /* Sub-Class ID */
    UINT16 deviceProtocol /* Protocol ID */
);
```

Refer to the Wind River USB for VxWorks API Reference for the API Definition for `usbdDynamicAttachRegister()`.

The following code example explains how keyboard class driver is registered with the translation unit:

```c
STATUS usbKeyboardDevInit (void)
{
    ...
    ...
    if (usbdClientRegister (KBD_CLIENT_NAME, &usbdHandle) != OK 
        || usbdDynamicAttachRegister (usbdHandle, 
            USB_CLASS_HID, 
            USB_SUBCLASS_HID_BOOT, 
            USB_PROTOCOL_HID_BOOT_KEYBOARD, FALSE, 
            usbKeyboardAttachCallback) 
        != OK)
```

74
The keyboard class driver un-registers itself with the lower layer by calling the routine `usbdClientUnregister()`.

```c
/*
 * un-registering the keyboard class driver
 */
usbdClientUnregister (usbdHandle);
usbdHandle = NULL;
```

### 4.3.4 Dynamic Device Attachment

Unlike most SIO drivers, the `usbKeyboardLib` driver’s number of supported channels is not fixed. Rather, USB keyboards can be added or removed from the system at any time. Thus, the number of channels is dynamic, and clients of `usbKeyboardLib` must be made aware of the appearance and disappearance of channels. Therefore, this driver includes a set of routines that allows clients to register for notification upon the attachment and removal of USB keyboards, and the corresponding creation and deletion of channels.

In order to be notified of the attachment and removal of USB keyboards, clients must register with `usbKeyboardLib` by calling `usbKeyboardDynamicAttachRegister()`. Clients provide a callback routine of the following form:

```c
typedef VOID (*USB_KBD_ATTACH_CALLBACK)(
    pVOID arg, /* caller-defined argument */
    SIO_CHAN *pChan, /* pointer to the affected SIO_CHAN */
    UINT16 attachCode /* defined as USB_KBD_xxxx */
);
```

When `usbKeyboardLib` detects a new USB keyboard, each registered callback is invoked with `pChan` pointing to a new `SIO_CHAN` structure and with `attachCode` set to the value `USB_KBD_ATTACH`. When keyboards are removed from the system, each registered callback is invoked with `attachCode` set to `USB_KBD_REMOVE` and with `pChan` pointing to the `SIO_CHAN` structure of the keyboard that has been removed.

The `usbKeyboardLib` driver maintains a usage count for each `SIO_CHAN` structure. Callers can increase the usage count by calling `usbKeyboardSioChanLock()`, or they can decrease it by calling
usbKeyboardSioChanUnlock(). Normally, if a keyboard is removed from the system when the usage count is zero (0), usbKeyboardLib automatically releases the SIO_CHAN structure formerly associated with the keyboard. However, clients that rely on this structure can use the locking mechanism to force the driver to retain the structure until it is no longer needed.

Consider an application that periodically polls a keyboard by calling the pollInput() callback identified in a particular SIO_CHAN structure. This is an example in which SIO_CHAN locking could be required.

The task responsible for polling runs in the background and operates asynchronously with respect to the code that receives attachment or detachment notification from usbKeyboardLib. If usbKeyboardLib frees the memory associated with the SIO_CHAN structure as soon as the keyboard is unplugged, it is possible that the pollInput() function pointer may be corrupted. If that occurs, the application’s asynchronous polling task would fail upon calling the, now corrupt, pollInput() function pointer, probably taking down the system.

The application can use the SIO_CHAN locking mechanism to force usbKeyboardLib to delay the release of the SIO_CHAN structure until after the application has canceled the background polling operation.

Example 4-1  SIO_CHAN Locking

The following code fragments demonstrate the typical use of the dynamic attachment and SIO_CHAN locking routines:

```c
/*************************************************************************
  * First, initialize the usbKeyboardLib.  
  */
  */
  if (usbKeyboardDevInit () != OK)
  {
    /* We failed to initialize usbKeyboardLib. */
    return ERROR;
  }

  /* Register for keyboard attachment/detachment notification. */
  if (usbKeyboardDynamicAttachRegister (kbdAttachCallback, (pVOID) 1234) != OK)
  {
    /* We failed to register for attachment notification. */
    return ERROR;
  }

  /* The kbdAttachCallback() routine will now be called asynchronously
   * whenever a keyboard is attached to or detached from the system. 
   */
  
  ...

  ...
```
Example 4-2  Keyboard Attachment

The keyboard attachment callback might resemble the following:

```c
/**************************************************************************
* kbdAttachCallback - receives callbacks from USB keyboard SIO driver
* RETURNS: N/A
*/
LOCAL SIO_CHAN *pOurChan = NULL;

LOCAL VOID kbdAttachCallback
(
    pVOID arg,  /* caller-defined argument */
    SIO_CHAN *pChan,  /* pointer to the affected SIO_CHAN */
    UINT16 attachCode  /* defined as USB_KBD_xxxx */
)
{
    UINT32 ourArg = (UINT32) arg;

    /* The argument is any arbitrary value that may be of use to this 
    * callback. In this example, we just demonstrate that the value 
    * originally passed to usbKeyboardDynamicAttachRegister() shows up here 
    * as our argument. */
    if (ourArg != 1234)
    {
        /* The argument never made it. */
        ...
    }

    switch (attachCode)
    {
        case USB_KBD_ATTACH:
            /* Lock the SIO_CHAN structure so it doesn’t disappear on us. */
            if (usbKeyboardSioChanLock (pChan) != OK)
            {
                /* This really shouldn’t be able to fail. */
                ...
            }

            /* Do other initialization stuff. */
            pOurChan = pChan;
            ...
            ...
            break;
```
case USB_KBD_DETACH:

    /* Tear down any data structures we may have created. */
    ...
    ...
    pOurChan = NULL;

    /* Allow usbKeyboardLib to release the SIO_CHAN structure. */
    usbKeyboardSioChanUnlock (pChan);
    break;
}
}

4.3.5  ioctl Routines

The usbKeyboardLib driver supports the SIO ioctl interface. However, attempts to set parameters, such as baud rates and start or stop bits, have no meaning in the USB environment and return ENOSYS.

4.3.6  Data Flow

For each USB keyboard connected to the system, usbKeyboardLib sets up a USB pipe to monitor keyboard input. Input, in the form of scan codes, is translated to ASCII codes and placed in an input queue. If SIO callbacks have been installed and usbKeyboardLib has been placed in the SIO interrupt mode of operation, usbKeyboardLib invokes the character received callback for each character in the queue. When usbKeyboardLib has been placed in polled mode, callbacks are not invoked and the caller must fetch keyboard input using the driver's pollInput() routine.

The usbKeyboardLib driver does not support output to the keyboard; therefore, calls to the txStartup() and pollOutput() routines return errors. The only output supported is the control of the keyboard LEDs, which usbKeyboardLib handles internally.

The caller must be aware that usbKeyboardLib is not capable of operating in a true polled mode, because the underlying USBD and USB HCD always operate in interrupt mode.

Example 4-3  Keyboard to Display Keystrokes

The following code fragment demonstrates using usbKeyboardLib to display keystrokes typed by the user:
int i;
char inChar;

/* Display the next ten keystrokes typed by the user. This code
* assumes that pOurChan was initialized as shown earlier and is
* currently not NULL.
*/

for (i = 0; i < 10; i++)
{
    /* Wait for a keystroke */
    while ((*pOurChan->pDrvFuncs->pollInput) (pOurChan, &inChar) != OK)
    {
    
        /* Display the keystroke. */
        printf ("The user pressed '%c'.\n", inChar);
    }

4.3.7 Typematic Repeat

USB keyboards do not implement typematic repeat, a feature that causes a key to
repeat if it is held down, typically for more than one-fourth or one-half second. If
you want this feature, implement it with the host software. For this purpose,
usbKeyboardLib creates a task, tUsbKbd, that monitors all open channels and
injects characters into input queues at an appropriate repeat rate.

For example, if a user presses and holds a key on a USB keyboard, a single report
is sent from the keyboard to the host indicating the keypress. If no report is
received within a pre-set interval indicating that the key has been released, the
tUsbKbd thread automatically injects additional copies of the same key into the
input queue at a pre-set rate. In the current implementation, the pre-set interval
(delay) is one-half (1/2) second, and the repeat rate is 15 characters per second.

4.3.8 Un-initializing the Keyboard Class Driver

The client application un-initializes the keyboard class driver by calling the routine
usbKeyboardDevShutdown (). This routine un-initializes the internal data
structures and removes the keyboard class driver from the USB Subsystem.

NOTE: The keyboard class driver maintains a usage count which is incremented
every time a client application tries to initialize the keyboard class driver. The
actual un-initialization of driver happens only when the usage count becomes 0.
Otherwise, for every call to usbKeyboardDevShutdown (), it is decremented by
one.
4.4 Mouse Driver

USB mice are described in human interface device (HID)-related specifications. The Wind River implementation of the USB mouse driver, `usbMouseLib`, concerns itself only with USB devices claiming to be mice as set forth in the USB specification; the driver ignores other types of HIDs, such as keyboards.

**NOTE:** USB mice operate according to either a boot protocol or a report protocol. However, the `usbMouseLib` driver enables mice for operation using the boot protocol only.

4.4.1 SIO Driver Model

The USB mouse class driver, `usbMouseLib`, follows the VxWorks serial I/O (SIO) driver model, with certain exceptions and extensions.

4.4.2 Initializing the Mouse Class Driver

`usbMouseLib` must be initialized by calling `usbMouseDevInit()`, which in turn initializes its connection to the USBD and other internal resources needed for operation. All interaction with the USB host controllers and devices is handled through the USBD.

Unlike some SIO drivers, `usbMouseLib` does not include data structures that require initialization prior to calling `usbMouseDevInit()`.

However, before a call to `usbMouseDevInit()`, the caller must ensure that the USBD has been properly initialized by calling, at a minimum, `usbdInitialize()`.

The caller must also confirm that at least one USB HCD is attached to the USBD, using `usbdHcdAttach()`, before mouse operation can begin; however, it is not necessary to call `usbdHcdAttach()` prior to initializing `usbMouseLib`. The `usbMouseLib` driver uses the USBD dynamic attachment services and recognizes USB mouse attachment and removal on the fly.

Unlike traditional SIO drivers, the `usbMouseLib` driver does not export entry points for send, receive, and error interrupts. All interrupt-driven behavior is managed by the underlying USBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of `usbMouseLib`. For the same reason, there is no post-interrupt-connect initialization code, and `usbMouseLib` therefore omits the `devInit2` entry point.
4.4.3 Registering the Mouse Class Driver

The mouse class driver registers with the USBD by calling `usbdClientRegister()`. In response to this call, the Translation Unit allocates mouse-client data structures and a mouse client callback task.

Once the mouse class driver is successfully registered with the translation unit, the translation unit returns a handle of type `USBD_CLIENT_HANDLE` which is used for all subsequent communication with the lower layers.

After the mouse class driver is successfully registered, the driver registers a callback routine with the lower layer by calling the routine `usbdDynamicAttachRegister()`. The callback is called whenever a device of particular class, subclass and protocol is connected to or disconnected from the USB Subsystem. The callback routine takes the following form:

```c
typedef VOID (*USBD_ATTACH_CALLBACK)(
    USBD_NODE_ID nodeId, /* USBD Handler */
    UINT16 attachAction, /* notification for device
                           * attachment or removal */
    UINT16 configuration, /* configuration index */
    UINT16 interface, /* interface number */
    UINT16 deviceClass, /* Class ID */
    UINT16 deviceSubClass, /* Sub-Class ID */
    UINT16 deviceProtocol/* Protocol ID */
);
```

Refer to the Wind River USB for VxWorks API Reference for the API Definition for `usbdDynamicAttachRegister()`.

The following code example explains how the mouse class driver is registered with the translation unit:

```c
/*
 * Registering the Mouse Class Driver
 */

STATUS usbMouseDevInit (void)
{
    …………..
    …………..
    if (usbdClientRegister (KBD_CLIENT_NAME, &usbdHandle) != OK
            || usbdDynamicAttachRegister (usbdHandle,
                USB_CLASS_HID,
                USB_SUBCLASS_HID_BOOT,
                USB_PROTOCOL_HID_BOOT_KEYBOARD, FALSE,
                usbMouseAttachCallback)
        != OK)
    …………..
```
The mouse class driver un-registers itself with the lower layer by calling the routine `usbdClientUnregister()`.

```c
/*
 * un-registering the mouse class driver
 */
usbdClientUnregister (usbdHandle);
usbdHandle = NULL;
```

### 4.4.4 Dynamic Device Attachment

As with the USB keyboard class driver, the number of channels supported by this driver is not fixed. Rather, USB mice can be added or removed from the system at any time. Thus, the number of channels is dynamic, and clients of `usbMouseLib` must be made aware of the appearance and disappearance of channels. Therefore, this driver includes a set of routines that allows clients to register for notification upon the attachment and removal of USB mice, and the corresponding creation and deletion of channels.

In order to be notified of the attachment and removal of USB mice, clients must register with `usbMouseLib` by calling `usbMouseDynamicAttachRegister()`. Clients provide a callback routine of the following form:

```c
typedef VOID (*USB_MSE_ATTACH_CALLBACK)
(   pVOID arg,       /* caller-defined argument */
   SIO_CHAN *pChan, /* pointer to the affected SIO_CHAN */
   UINT16 attachCode /* defined as USB_MSE_xxxx */
);
```

When `usbMouseLib` detects a new USB mouse, each registered callback is invoked with `pChan` pointing to a new `SIO_CHAN` structure and with `attachCode` set to the value `USB_MSE_ATTACH`. When a mouse is removed from the system, each registered callback is invoked with `attachCode` set to `USB_MSE_REMOVE` and with `pChan` pointing to the `SIO_CHAN` structure of the mouse that has been removed.

As with `usbKeyboardLib`, `usbMouseLib` maintains a usage count for each `SIO_CHAN` structure. Callers can increment the usage count by calling `usbMouseSioChanLock()` and can decrement it by calling `usbMouseSioChanUnlock()`. For more information on using the `SIO_CHAN` structure, see Example 4-1 in 4.3.4 Dynamic Device Attachment, p.75.
4.4.5 ioctl Routines

The usbMouseLib driver supports the SIO ioctl interface. However, attempts to set parameters, such as baud rates and start or stop bits, have no meaning in the USB environment and return ENOSYS.

4.4.6 Data Flow

For each USB mouse connected to the system, usbMouseLib sets up a USB pipe to monitor input from the mouse, in the form of HID boot reports. These mouse boot reports are of the following form (as defined in usbHid.h):

```c
typedef struct hid_mse_boot_report {
    UINT8 buttonState;  /* buttons */
    char xDisplacement;  /* signed x-displacement */
    char yDisplacement;  /* signed y-displacement */
} HID_MSE_BOOT_REPORT, *pHID_MSE_BOOT_REPORT;
```

In order to receive these reports, a client of usbMouseLib must install a special callback using the driver’s callbackInstall( ) routine. The callback type is SIO_CALLBACK_PUT_MOUSE_REPORT, and the callback itself takes the following form:

```c
VOID (*MSE_REPORT_CALLBACK) (void *callbackArg, /* caller-defined argument */
    pHID_MSE_BOOT_REPORT pReport /* pointer to mouse boot report */);
```

The client callback interprets the report according to its needs, saving any data that might be required from the HID_MSE_BOOT_REPORT structure.

The usbMouseLib driver does not support polled modes of operation; nor does it support the traditional SIO_CALLBACK_PUT_RCV_CHAR callback. Given the structured nature of the boot reports received from USB mice, character-by-character input of boot reports would be inefficient and could lead to report framing problems in the input stream.

4.4.7 Un-initializing the Mouse Class Driver

The client application un-initializes the mouse class driver by calling the routine usbMouseDevShutdown(). This routine un-initializes the internal data structures and removes the mouse class driver from the USB Subsystem.
4.5 Printer Driver

USB printers are described in human interface device (HID)-related specifications. The printer class driver specification presents two kinds of printers: uni-directional printers (output only) and bi-directional printers (capable of both output and input). The `usbPrinterLib` driver is capable of handling both types of printers. If a printer is uni-directional, the driver only allows characters to be written to the printer; if the printer is bi-directional, it allows both output and input streams to be written and read.

4.5.1 SIO Driver Model

The USB printer class driver, `usbPrinterLib`, follows the VxWorks serial I/O (SIO) driver model, with certain exceptions and extensions. This driver provides the external APIs expected of a standard multi-mode serial (SIO) driver and adds extensions that support the hot-plugging USB environment.

4.5.2 Initializing the Printer Driver

As with standard SIO drivers, `usbPrinterLib` must be initialized by calling `usbPrinterDevInit()`, which in turn initializes its connection to the USBD and other internal resources needed for operation. All interaction with the USB host controllers and devices is handled through the USBD.

Unlike some SIO drivers, `usbPrinterLib` does not include data structures that require initialization prior to calling `usbPrinterDevInit()`.

However, before a call to `usbPrinterDevInit()`, the caller must ensure that the USBD has been properly initialized. For details for how to initialize the USB host stack, refer to 2.5 Initializing the USB Host Stack Hardware, p.23.

Unlike traditional SIO drivers, the `usbPrinterLib` driver does not export entry points for send, receive, and error interrupts. All interrupt-driven behavior is managed by the underlying USBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of `usbPrinterLib`. For the same reason, there is no post-interrupt-connect initialization code, and `usbPrinterLib` therefore omits the `devInit2` entry point.
4.5.3 Registering the Printer Driver

The printer class driver registers with the USBD by calling `usbdClientRegister()`. In response to this call, the Translation Unit allocates printer-client data structures and a printer client callback task.

Once the printer class driver is successfully registered with the translation unit, the translation unit returns a handle of type `USBD_CLIENT_HANDLE` which is used for all subsequent communication with the lower layers.

After the printer class driver is successfully registered, the driver registers a callback routine with the lower layer by calling the routine `usbdDynamicAttachRegister()`. The callback is called whenever a device of particular class, subclass and protocol is connected to or disconnected from the USB Subsystem. The callback routine takes the following form:

```c
typedef VOID (*USBD_ATTACH_CALLBACK)
    (USBD_NODE_ID nodeId,  /* USBD Handler */
     UINT16 attachAction, /* notification for device
                      * attachment or removal
     * USB_DYNA_ATTACH or
     * USB_DYNA_REMOVE */
     UINT16 configuration,/* configuration index */
     UINT16 interface,/* interface number */
     UINT16 deviceClass, /* Class ID */
     UINT16 deviceSubClass, /* Sub-Class ID */
     UINT16 deviceProtocol/* Protocol ID */);
```

Refer to the *Wind River USB for VxWorks API Reference* for the API definition for `usbdDynamicAttachRegister()`.

The following code example explains how the printer class driver is registered with the translation unit:

```c
/*
 * Registering the printer class driver
 */

STATUS usbPrinterDevInit (void)
{
    ..........;

    if (usbdClientRegister (PRN_CLIENT_NAME, &usbdHandle) != OK
        || usbdDynamicAttachRegister (usbdHandle,
                                      USB_CLASS_PRINTER,
                                      USB_SUBCLASS_PRINTER,
                                      USBD_NOTIFY_ALL, FALSE,
                                      .....
```
The printer class driver un-registers itself with the lower layer by calling the routine \texttt{usbdClientUnregister()}. 

```
/* un-registering the printer class driver */
usbdClientUnregister (usbdHandle);
```

4.5.4 Dynamic Device Attachment

As with the USB keyboard class driver, the number of channels supported by this driver is not fixed. Rather, USB printers can be added or removed from the system at any time. Thus, the number of channels is dynamic, and clients of \texttt{usbPrinterLib} must be made aware of the appearance and disappearance of channels. Therefore, this driver includes a set of routines that allows clients to register for notification upon the attachment and removal of USB printers, and the corresponding creation and deletion of channels.

In order to be notified of the attachment and removal of USB printers, clients must register with \texttt{usbPrinterLib} through the routine \texttt{usbPrinterDynamicAttachRegister()}. Clients provide a callback routine of the following form:

```
typedef VOID (*USB_PRN_ATTACH_CALLBACK)(
    pVOID arg, /* caller-defined argument */
    SIO_CHAN *pChan, /* pointer to the affected SIO_CHAN */
    UINT16 attachCode /* defined as USB_PRN_xxxx */
);
```

When \texttt{usbPrinterLib} detects a new USB printer, each registered callback is invoked with \texttt{pChan} pointing to a new \texttt{SIO_CHAN} structure and with \texttt{attachCode} set to the value \texttt{USB_PRN_ATTACH}. When printers are removed from the system, each registered callback is invoked with \texttt{attachCode} set to \texttt{USB_PRN_REMOVE} and with \texttt{pChan} pointing to the \texttt{SIO_CHAN} structure of the printer that has been removed.

As with \texttt{usbKeyboardLib}, \texttt{usbPrinterLib} maintains a usage count for each \texttt{SIO_CHAN} structure. Callers can increment the usage count by calling
usbPrinterSioChanLock() and can decrement it by calling usbPrinterSioChanUnlock(). Normally, if a printer is removed from the system when the usage count is zero (0), usbPrinterLib automatically releases the SIO_CHAN structure formerly associated with the printer. However, clients that rely on this structure can use this locking mechanism to force the driver to retain the structure until it is no longer needed. For more information about using the SIO_CHAN structure, see Example 4-1 in 4.3.4 Dynamic Device Attachment, p.75.

4.5.5 ioctl Routines

The usbPrinterLib driver supports the SIO ioctl interface. However, attempts to set parameters, such as baud rates and start or stop bits, have no meaning in the USB environment and are treated as no-ops.

Additional ioctl routines have been added to allow the caller to retrieve the USB printer's device ID string, the type of printer (uni- or bi-directional), and the current printer status. The device ID string is discussed in more detail in the USB specification and is based on the IEEE-1284 device ID string used by most 1284-compliant printers. The printer status routine can be used to determine whether the printer has been selected, is out of paper, or has an error condition.

4.5.6 Data Flow

For each USB printer connected to the system, usbPrinterLib sets up a USB pipe to output bulk data to the printer. This is the pipe through which printer control and page description data are sent to the printer. Additionally, if the printer is bi-directional, usbPrinterLib sets up a USB pipe to receive bulk input data from the printer. The meaning of data received from a bi-directional printer depends on the particular printer make and model.

The USB printer driver supports only SIO_MODE_INT, the SIO interrupt mode of operation. Any attempt to place the driver in polled mode returns an error.
4.6 Speaker Driver

USB speakers are described in the *USB Device Class Definition for Audio Devices*. The Wind River implementation of the USB speaker driver, `usbSpeakerLib`, supports only USB speakers as defined by this specification and ignores other types of USB audio devices, such as MPEG and MIDI devices.

**NOTE:** Some models of USB speakers are implemented as compound devices, which often integrate a small number of physical audio controls, such as those for volume, bass, treble, and balance. These physical controls are presented as separate USB interfaces within the compound device and are implemented according to the human interface device (HID) specification.

The `usbSpeakerLib` library ignores these non-audio interfaces. If the target application requires these HID controls to be enabled, you must implement additional logic to recognize the HID interface and map the HID routines to appropriate `usbSpeakerLib` ioctl routines (see 4.6.6 ioctl Routines, p. 91).

4.6.1 SEQ_DEV Driver Model

The `usbSpeakerLib` driver provides a modified VxWorks SEQ_DEV interface to its callers. Among existing VxWorks driver models, the SEQ_DEV interface best supports the streaming data transfer model required by isochronous devices such as USB speakers. As with other VxWorks USB class drivers, the standard driver interface has been expanded to support features unique to the USB and to speakers in general. Routines have been added to allow callers to recognize the dynamic attachment and removal of speaker devices. ioctl routines have been added to retrieve and control additional settings related to speaker operation.

4.6.2 Initializing the Speaker Driver

As with standard SEQ_DEV drivers, this driver must be initialized by calling `usbSpeakerDevInit()`. The `usbSpeakerDevInit()` routine, in turn, initializes its connection to the USBD and other internal resources needed for operation.

However, before a call to `usbSpeakerDevInit()`, the caller must ensure that the USBD has been properly initialized. Please refer to 2.5 Initializing the USB Host Stack Hardware, p. 23 for how to initialize USB host stack.
The USB Speaker Library used the USBD dynamic attachment services to recognize the USB Speaker attachment and detachment on the fly.

### 4.6.3 Registering the Speaker Driver

The speaker class driver registers with the USBD by calling `usbdClientRegister()`.

In response to this call, the Translation Unit allocates speaker-client data structures and a speaker-client callback task.

Once the speaker class driver is successfully registered with the translation unit, the translation unit returns a handle of type `USBD_CLIENT_HANDLE` which is used for all subsequent communication with the lower layers.

After the speaker class driver is successfully registered, the driver registers a callback routine with the lower layer by calling the routine `usbdDynamicAttachRegister()`. The callback is called whenever a device of particular class, subclass and protocol is connected to or disconnected from the USB Subsystem. The callback routine takes the following form:

```c
typedef VOID (*USBD_ATTACH_CALLBACK)(
    USBD_NODE_ID nodeId, /* USBD Handler */
    UINT16 attachAction, /* notification for device attachment or removal */
    * USBD_DYNA_ATTACH or
    * USBD_DYNA_REMOVE */
    UINT16 configuration, /* configuration index */
    UINT16 interface, /* interface number */
    UINT16 deviceClass, /* Class ID */
    UINT16 deviceSubClass, /* Sub-Class ID */
    UINT16 deviceProtocol /* Protocol ID */
);
```

Please refer to the Wind River USB for VxWorks API Reference for the API definition for `usbdDynamicAttachRegister()`.

The following code example explains how the speaker class driver is registered with the translation unit:

```c
/* Registering the speaker Class Driver */

STATUS usbSpeakerDevInit (void)
{
    ...................
    ...................
    if (usbdClientRegister (SPKR_CLIENT_NAME, &usbdHandle) != OK
        || usbdDynamicAttachRegister (usbdHandle,
            USB_CLASS_PRINTER,
```

89
The speaker class driver un-registers itself with the lower layer by calling the routine `usbdClientUnregister()`.

```c
/*
 * un-registering the speaker class driver
 */
usbdClientUnregister (usbdHandle);
usbdHandle = NULL;
```

### 4.6.4 Dynamic Device Attachment

Like other USB devices, USB speakers can be attached to or detached from the system dynamically. The `usbSpeakerLib` driver uses the USBD's dynamic attachment services to recognize these events, and callers to `usbSpeakerLib` can use the `usbSpeakerDynamicAttachRegister()` routine to register with the driver to be notified when USB speakers are attached or removed. The caller must provide `usbSpeakerDynamicAttachRegister()` with a pointer to a callback routine of the following form:

```c
typedef VOID (*USB_SPKR_ATTACH_CALLBACK)(
    pVOID arg,    /* caller-defined argument */
    SEQ_DEV *pSeqDev, /* pointer to the affected SEQ_DEV */
    UINT16 attachCode /* defined as USB_SPKR_xxxx */
);
```

When a USB speaker is attached or removed, `usbSpeakerLib` invokes each registered notification callback. The callback is passed a pointer to the affected SEQ_DEV structure, `pSeqDev`, and an attachment code, `attachCode`, indicating whether the speaker is being attached (USB_SPKR_ATTACH) or removed (USB_SPKR_REMOVE).

The `usbSpeakerLib` driver maintains a usage count for each SEQ_DEV structure. Callers can increment the usage count by calling `usbSpeakerSeqDevLock()` or can decrement the count by calling `usbSpeakerSeqDevUnlock()`. If a USB speaker is removed from the system when its usage count is 0, `usbSpeakerLib` automatically removes all data structures, including the SEQ_DEV structure itself,
that have been allocated on behalf of the device. However, callers sometimes rely on these data structures and must properly recognize the removal of the device before it is safe to destroy the underlying data structures. The lock and unlock routines provide a mechanism for callers to protect these data structures if required.

4.6.5 Recognizing and Handling USB Speakers

Speakers, loosely defined, are USB audio devices that provide an output terminal. For each USB audio device, `usbSpeakerLib` examines the descriptors that define both the units and terminals contained within the device and how they are connected.

If an output terminal is found, `usbSpeakerLib` traces the device's internal connections to determine which input terminal provides the audio stream for that output terminal, and which feature unit, if any, is responsible for controlling audio stream attributes such as volume. After building an internal map of the device, `usbSpeakerLib` configures the device and waits for a caller to provide a stream of audio data. If no output terminal is found, `usbSpeakerLib` ignores the audio device.

After determining that the audio device contains an output terminal, `usbSpeakerLib` builds a list of the audio formats that the device supports. The `usbSpeakerLib` driver supports only “AudioStreaming” interfaces (no “MIDIStreaming” is supported). For each USB speaker attached to the system and properly recognized by `usbSpeakerLib`, `usbSpeakerLib` creates a `SEQ_DEV` structure to control the speaker. Each speaker is uniquely identified by the pointer to its corresponding `SEQ_DEV` structure.

4.6.6 ioctl Routines

The `usbSpeakerLib` driver implements a number of ioctl routines unique to the handling of audio data and devices. The driver uses ioctl routines to set the mute, volume, bass, mid-range, and treble controls. The driver provides ioctl routines for use by callers to interrogate a speaker's audio format capabilities or to specify the audio format for a subsequent data stream. The driver also provides ioctl routines to mark the beginning and end of audio data streams (see 4.6.7 Data Flow, p.92).
4.6.7 Data Flow

Before sending audio data to a speaker device, the caller must specify the data format (for example, PCM or MPEG) using an ioctl routine (see 4.6.6 ioctl Routines, p.91). The USB speaker itself must support the specified data format or a similar one.

USB speakers rely on an uninterrupted, time-critical stream of audio data. The data is sent to the speaker through an isochronous pipe. In order for the data flow to continue uninterrupted, usbSpeakerLib uses an internal double-buffering scheme. When the caller presents data to the usbSpeakerLib sd_seqWrt() routine, usbSpeakerLib copies the data into an internal buffer and immediately releases the caller’s buffer. The caller immediately tries to pass the next buffer to usbSpeakerLib. When the usbSpeakerLib internal buffer is filled, it blocks the caller until it can accept new data. In this manner, the caller and usbSpeakerLib work together to ensure that an adequate supply of audio data is always available to continue an uninterrupted isochronous transmission.

Audio play begins after usbSpeakerLib has accepted a half-second of audio data or when the caller closes the audio stream, whichever happens first. The caller must use ioctl routines to open and close each audio stream. The usbSpeakerLib driver relies on these open and close ioctl routines to manage its internal buffers correctly.
4.7 Mass Storage Class Driver

USB mass storage class devices are described in the mass storage class specification and can behave according to several different implementations. Wind River supplies drivers that adhere to the Bulk-Only and Control/Bulk/Interrupt (CBI) implementation methods. Each of these two drivers uses a command set from an existing protocol. The Wind River Bulk-Only driver wraps USB protocol around the commands documented in *SCSI Primary Commands: 2 (SPC-2), Revision 3* or later. The Wind River CBI driver wraps USB protocol around the commands documented in the *USB Floppy Interface (UFI) Command Specification*.


A device’s subclass code, presented in its interface descriptor, indicates which of these command sets the device understands. Table 4-1, adapted from the *Universal Serial Bus Mass Storage Class Specification Overview*, shows the command set that corresponds to each subclass code.

<table>
<thead>
<tr>
<th>Subclass Code</th>
<th>Command Block Specification</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>01h</td>
<td>Reduced Block Commands (RBC) T10 Project 1240-D</td>
<td>Typically, a flash device uses RBCs. However, any mass storage device can use RBCs.</td>
</tr>
<tr>
<td>02h</td>
<td>SFF-8020i or MMC-2 (ATAPI)</td>
<td>Typically, a CD/DVD device uses SFF-8020i or MMC-2 command blocks for its mass storage interface.</td>
</tr>
<tr>
<td>03h</td>
<td>QIC-157</td>
<td>Typically, a tape device uses QIC-157 command blocks.</td>
</tr>
<tr>
<td>04h</td>
<td>UFI</td>
<td>Typically, a floppy disk drive (FDD) device uses UFI command blocks.</td>
</tr>
</tbody>
</table>
Wind River’s CBI driver responds to devices with subclass code 04h. Wind River’s Bulk-Only driver responds to devices with subclass code 06h.

All references to the mass storage class driver library take the form `usbMSCxxx()`. References to Wind River’s CBI driver take the form `usbCbiUfixxx()`; references to Wind River’s Bulk-Only driver take the form `usbBulkxxx()`.

### 4.7.1 Extended Block Device Driver Model

A mass storage class driver is a type of block device driver that provides generic direct access to a block device through VxWorks. Mass storage class drivers interact with the file system. The file system, in turn, interacts with the I/O system.

The Extended Block Device (XBD) uses the Event Reporting Framework (ERF) and the device infrastructure to interface to drivers and higher levels of functionality. An XBD structure is allocated by the driver that needs to use that interface.

<table>
<thead>
<tr>
<th>Subclass Code</th>
<th>Command Block Specification</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>05h</td>
<td>SFF-8070i</td>
<td>Typically, a floppy disk drive (FDD) device uses SFF-8070i command blocks. However, an FDD device can belong to another subclass (for example, RBC); likewise, other types of storage device can belong to the SFF-8070i subclass.</td>
</tr>
<tr>
<td>06h</td>
<td>SCSI transparent command set</td>
<td></td>
</tr>
<tr>
<td>07h-FFh</td>
<td>Reserved for future use</td>
<td></td>
</tr>
</tbody>
</table>
Figure 4-5  USB Block Driver Hierarchy in a VxWorks System

Application

I/O System

Non-Block Device Driver

Event Reporting Framework

File System 1

File System 2

Partitioning Layer

Part 1 Part 2 Part 3 Part 4

USB Mass Storage Class (XBD) Driver

USBD Host Driver

USB Host Controller

USB Devices

VxWorks USB Host Stack

4 USB Class Drivers
4.7 Mass Storage Class Driver
With respect to the USB mass storage block device driver, when it is determined that a device insertion has taken place, the `usbMSCBlkDevCreate()` allocates an XBD for that device. An insertion event is generated for the ERF which propagates that insertion event to any higher level function that might be waiting for a device insertion. The XBD consists of two main components:

- **XBD structure**

  The XBD structure defines the XBD, which keeps the device entry that makes this an insertable/removable interface, function pointers for the methods of this device, device block size and number of blocks information. The XBD structure is defined in `target/h/drv/xbd/xbd.h`.

- **BIO structure**

  The Block_IO (BIO) structure contains the necessary information to read or write to the block device, or to another XBD. The BIO structure is defined in `target/h/drv/xbd/bio.h`.

For each USB device, a `tUsbMSCXbdStrategy` task is spawned to process the BIOS passed down from the file system through `usbMSCXbdStrategy`. If a device has a multiple logical unit, then a `tUsbMSCXbdStrategy` task is spawned for each LUN.

The shaded area of Figure 4-5 illustrates the following sequence of events:

1. File system 1 and file system 2 register with the ERF looking for the device insertion event.
2. The device is inserted and the device driver notifies the ERF of the insertion. The driver also creates an XBD interface for the mass storage device.
3. The ERF reports the insertion to the registered file systems, and the file systems can now read the device.
4. The file systems create a partitioning layer on top the device driver XBD to make use of the partitions that exist on the disk.

### 4.7.2 API Routines

The USB mass storage class device driver provides the following API routines to the file system:

- **`usbMSCDevInit()`**

  This routine is a general initialization routine. It performs all operations that are to be done one time only. It initializes required data structures and registers the mass storage class driver with the USBD. It also registers for notification
when mass storage devices are dynamically attached. Traditionally, this routine is called from the VxWorks boot code.

**usbMSCDevCreate()**

This routine creates a logical block device structure for a particular USB mass storage device. At least one mass storage device must exist on the USB when this routine is invoked.

**usbMSCBlkWrt()**

This routine writes to a specified physical block or blocks from a specified USB device.

**usbMSCBlkRd()**

This routine reads a specified physical block or blocks from a specified USB device.

**usbMSCDevIoctl()**

This routine performs any device-specific I/O control routines. For example, it sends commands that are not explicitly used by a typical file system, and it sets device configuration.

**usbMSCStatusChk()**

This routine checks for the status of the USB device. This is primarily for removable media such as USB floppy drives. A change in status is reported to the file system mounted, if any.

### 4.7.3 Dynamic Attachment

A USB device driver must support the most important feature of a USB device: dynamic insertion and removal of the device. A USB mass storage class device can be plugged into and out of the system at any time. This hotswap feature is supported by a callback mechanism.

The USB mass storage class driver provides the registration routine **usbMSCDynamicAttachRegister()**, which registers the client with the driver. When a USB mass storage class device is attached to or removed from the system, all clients are notified through the USBD’s call to a user-provided callback routine. The callback routine receives the **USB_NODE_ID** of the attached device and a flag that is set to either **USB_MSC_ATTACH** or **USB_MSC_DETACH**, indicating the attachment or removal of the device. The driver also provides the **usbMSCDynamicAttachUnregister()** routine for de-registering a block device driver.
The mass storage class driver maintains a usage count of XBD structures. When a client uses a XBD structure, it informs the driver by calling `usbMSCDevLock()`. For each `usbMSCDevLock()` call, the driver increments the usage count. When a client is finished with the XBD structure, it must notify the driver by calling `usbMSCDevUnlock()`. The driver then decrements the usage count. Normally, if a mass storage class device is removed from the system when the usage count is zero (0), the driver releases the corresponding XBD structure. However, clients that rely on this structure can use this locking mechanism to force the driver to retain the structure until it is no longer needed.

4.7.4 Initialization

The mass storage class driver is initialized through the `usbMSCDevInit()` routine. This API call, in turn, initializes internal resources needed for its operation and registers a callback routine with the USBD. The callback routine is then invoked whenever a USB mass storage class device is attached to or removed from the system.

All interactions between the USB host controller and the mass storage class device are handled through the USBD. Therefore, before calling `usbMSCDevInit()`, the user must ensure that the USBD has been properly initialized with `usbdInitialize()`. Also, before any operation with a block device driver, the caller must ensure that at least one host controller is attached to the USBD.

4.7.5 Data Flow

The mass storage class driver’s data read and write mechanism behaves like that of a standard block device driver. It uses the data read and write function pointers that are installed through the `usbMSCBlkDevCreate()` routine. Because most USB mass storage class devices can implement 64-byte endpoints only, the driver must manage the transfer of the larger chunks (that is, 512-byte blocks) of data that are understood by the file system. To facilitate the multiple read/write transactions that are necessary to complete the block access, the USBD uses its IRP mechanism. This allows the user to specify a routine, `usbMSCIrpmCallback()`, that is called when the block transaction is complete.
4.8 Communication Class Drivers

USB communication class devices can behave according to several different implementations. Wind River supplies drivers for Ethernet networking control model devices.

4.8.1 Ethernet Networking Control Model Driver

This section describes Wind River’s USB networking control model driver. This driver supports USB Ethernet networking control model devices (network adapters) with subclass code 06h (see Table 4-1), with certain exceptions and extensions.

The networking control model driver presents two interfaces for transferring information to a device: a communication class interface and a data class interface. The communication class interface is a management interface and is required of all communication devices. The data class interface can be used to transport data across the USB wire. Ethernet data frames are encapsulated into USB packets and are then transferred using this data class interface. These Ethernet packets include an Ethernet destination address (DA), which is appended to the data field. Ethernet packets in either direction over the USB do not include a CRC (cyclic redundancy check); error-checking is instead performed on the surrounding USB packet.

The hierarchy diagram in Figure 4-6 illustrates where the USB communication class driver fits into a VxWorks system.

4.8.2 Enhanced Network Driver Model

Wind River’s USB networking control model driver conforms to the MUX Enhanced Network Driver (END) model, with certain variations. These differences are designed to accommodate the following features:

- Hotswap of USB devices.
- Attachment of multiple identical devices to one host.
- Support for USB network devices with vendor-specific initialization requirements. For example, the KSLI adapter requires that new firmware be downloaded before normal operation begins.
- A dynamic insertion and removal callback mechanism.
In order to meet these requirements, Wind River drivers include additional APIs beyond those defined in the standard END specification. For detailed information on the END model, see the VxWorks BSP Developer’s Guide, 5.5 or the VxWorks Device Driver Developer’s Guide, 6.0.

Figure 4-6  USB Communication Class Driver Hierarchy in a VxWorks System
4.8.3 Dynamic Attachment

Because USB network adapters can be hotswapped to and from the system at any time, the number of devices is dynamic. Clients of usbXXXEndLib (where XXX specifies the supported device) can be made aware of the attachment and removal of devices. The network control model driver includes a set of API calls that allows clients to register for notification upon attachment or removal of a USB network adapter device. The attachment and removal of USB network adapters correspond, respectively, to the creation and deletion of USB_XXX_DEV structures.

In order to be notified of the attachment or removal of USB network adapters, clients must register with usbXXXEndLib by calling usbXXXDynamicAttachRegister(), providing a callback routine—for example, pegasusAttachCallback().

When usbXXXEndLib detects a new USB network adapter, each registered client callback routine is invoked with callbackType set to USB_XXX_ATTACH. Similarly, when a USB network adapter is removed from the system, each registered client callback routine is invoked with callbackType set to USB_XXX_DETACH.

The usbXXXEndLib driver maintains a usage count for each USB_XXX_DEV structure. When a client uses the USB_XXX_DEV structure, it informs the driver by calling usbXXXDevLock(). For each usbXXXDevLock() call, the driver increments the usage count. When a client is finished with the USB_XXX_DEV structure, it must notify the driver by calling usbXXXDevUnlock(). The driver then decrements the usage count. Normally, if an adapter is removed from the system when the usage count is zero (0), the driver releases the corresponding USB_XXX_DEV structure. However, clients that rely on this structure can use this locking mechanism to force the driver to retain the structure until it is no longer needed.

4.8.4 Initialization

The usbXXXEndLib driver must be initialized through the usbXXXEndInit() routine, which in turn initializes its connection to the USBD and other internal resources needed for its operation. This API call also registers a callback routine with the USBD. The callback routine is then invoked whenever a USB networking device is attached to or removed from the system.

All interactions between the USB host controller and the networking device are handled through the USBD. Therefore, before calling usbXXXEndInit(), the user must ensure that the USBD has been properly initialized with usbdInitialize(). Also, before any operation with a networking device driver, the caller must ensure
that at least one host controller has been attached to the USBD through 
\texttt{usbUhcdInit()}, \texttt{usbOhciInit()}, or \texttt{usbEhcdInit()}.

### Interrupt Behavior

The \texttt{usbXXXEndLib} driver relies on the underlying USBD and HCD layers to communicate with USB Ethernet networking control model devices (network adapters). The USBD and HCD layers, in turn, use the host controller interrupt for this communication. In this way, all interrupt-driven behavior is managed by the underlying USBD and HCD layers. Therefore, there is no need for the caller (or BSP) to connect interrupts on behalf of \texttt{usbXXXEndLib}. For the same reason, there is no post-interrupt-connect initialization code and \texttt{usbXXXEndLib} omits the \texttt{devInit2} entry point.

The \texttt{usbXXXEndLib} driver inherently depends on the host controller interrupt for its communication with USB network adapters. Therefore, the driver supports only the interrupt mode of operation. Any attempt to place the driver in the polled mode returns an error.

### ioctl Routines

The \texttt{usbXXXEndLib} driver supports the END ioctl interface. However, any attempt to place the driver in polled mode returns an error.

### Data Flow

The \texttt{usbXXXEndLib} driver’s data transmission mechanism deviates slightly from the END standard in that all data is routed through the USBD. The \texttt{XXXEndSend()} routine fills in the \texttt{pUSB_IRP} structure and exports the structure to the USBD to send or receive the data.

IRPs are a mechanism for scheduling data transfers across the USB. For example, for the host to receive data from a network device, an IRP using the bulk input pipe is formatted and submitted to the USBD by the driver. When data becomes available, \texttt{XXXEndRecv()} is invoked by the IRP’s callback to process the incoming packet.

Whenever data is transferred through the USBD using the \texttt{pUSB_IRP} structure, callback routines are passed to the USBD. These callback routines acknowledge the transmission of each packet of data. The execution of each callback routine indicates that the corresponding data packet has been successfully transmitted.
5

USB Peripheral Stack
Target Layer Overview

5.1 Introduction 103
5.2 Initializing the Target Layer 105
5.3 Attaching and Detaching a TCD 105
5.4 Enabling and Disabling the TCD 110
5.5 Implementing Target Application Callback Routines 112
5.6 Pipe-Specific Requests 126
5.7 Device Control and Status Information 137
5.8 Shutdown Procedures 138

5.1 Introduction

This chapter shows how to create a target application that interfaces with the target layer in the Wind River USB peripheral stack. For instructions on how to create a target controller driver that interfaces with the Hardware Adaptation Layer in the Wind River USB peripheral stack, see 6. Target Controller Drivers.

Figure 5-1 shows the target layer and the various interfaces exposed by this layer to the target application and the HAL, and how these layers and subsystems interact.
To communicate through the target layer, the application in the configlette and your target application must coordinate to:

- initialize the target layer (see 5.2 Initializing the Target Layer, p.105)
- implement certain callback routines (see 5.5 Implementing Target Application Callback Routines, p.112)
- attach to a TCD (see 5.3 Attaching and Detaching a TCD, p.105)
- enable the TCD (see 5.4 Enabling and Disabling the TCD, p.110)
- create pipes (see 5.6.1 Creating and Destroying the Pipes, p.127)
- and transfer data (see 5.6.2 Transferring and Aborting Data, p.130)

**NOTE:** The target application should implement all control requests, as specified in Section 9.4 Standard Device Request of the USB2.0 Specification document.
5.2 Initializing the Target Layer

An application in the configlette initializes the target layer in a two-stage process:

1. It calls `usbTargInitialize()` at least once. The `usbTargInitialize()` routine (which requires no parameters) initializes internal target layer data structures. You may call `usbTargInitialize()` once or many times. The target layer maintains a usage count that is incremented for each successful call to `usbTargInitialize()` and decremented for each corresponding call to `usbTargShutdown()` (a routine that also requires no parameters). The target layer only truly initializes itself when the usage count goes from zero to one, and it only truly shuts down when that usage count returns to zero.

2. It attaches the target application to at least one TCD with the `usbTargTcdAttach()` routine.

3. Once the TCD is attached, the target application should enable the TCD by calling `usbTargEnable()`.

5.3 Attaching and Detaching a TCD

Before the target application can receive and respond to the requests from the host, the application in the configlette must attach it to the TCD by using `usbTargTcdAttach()`. The application in the configlette passes the following arguments to `usbTargTcdAttach()`:

- the TCD single entry point (see 6.3 Single Entry Point, p.142)
- a pointer to the TCD-defined parameters
- a pointer to the target application callback table (see Target Application Callback Table, p.108)
- a parameter (`callbackParam`), the nature of which is up to the target application, that the target application wants to receive in these callback routines

Once the target controller successfully attaches itself with the TCD, the TCD returns a handle to itself. The handle is stored in `USB_TARG_CHANNEL`. The target application uses this handle for all subsequent communication with the TCD.
After the TCD is attached, the target application should enable the TCD by calling `usbTargEnable()`. See 5.4 Enabling and Disabling the TCD, p.110 for the interface definition for `usbTargEnable()`.

**TCD-defined Parameters**

The TCD-defined parameters are specific to the particular TCD. You can hard-code these if you know beforehand the implementation details of the TCD your application will be using, or you may be able to retrieve the values for these parameters by calling the PCI interfaces for the device. There are configlette routines available in `/target/config/comps/src/usrUsbTargPciInit.c` file. The target application must call these configlette routines to populate the TCD-defined parameters. For more information, see 2.6 Initializing the USB Peripheral Stack Hardware, p.28.

**NOTE:** The target application must fill the TCD-defined parameters prior to attaching the TCD with the target application.

**Example 5-1 Example Retrieving Configuration Parameters**

This example demonstrates how to find the ISP 1582 target controller with the `pciFindDevice()` routine, and retrieve its configuration parameters with `usbPciConfigHeaderGet()`:

```c
bStatus = pciFindDevice( ISP1582_VENDOR_ID,
ISP1582_DEVICE_ID,
nDeviceIndex,
&PCIBusNumber,
&PCIDeviceNumber,
&PCIFunctionNumber );

/* Check whether the isp1582 Controller was found */
{
   if( bStatus != OK )
   {
      /* No ISP1592 Device found */
      printf( "pciFindClass returned error \n " );
      return;
   }

   /* Get the configuration header */

   usbPciConfigHeaderGet( PCIBusNumber,
   PCIDeviceNumber,
   PCIFunctionNumber,
   &pciCfgHdr );
```

106
The structure of the parameters depends on the particular TCD in use, but can take a form like those shown below. For PDIUSB12, it can take the form:

```c
typedef struct usbTcdPdiusbd12Params /* USB_TCD_PDIUSBD12_PARAMS */
{
    UINT32 ioBase; /* Base I/O address range */
    UINT16 irq; /* IRQ channel (e.g., 5 = IRQ5) */
    UINT16 dma; /* DMA channel (e.g., 3 = DMA3) */
} USB_TCD_PDIUSBD12_PARAMS, *pUSB_TCD_PDIUSBD12_PARAMS;
```

For NET2280, it can take the form:

```c
typedef struct usbTcdNET2280Params
{
    UINT32 ioBase[NET2280_NO_OF_PCI_BADDR]; /* IO base array */
    UINT8 irq; /* IRQ value */
} USB_TCD_NET2280_PARAMS, *pUSB_TCD_NET2280_PARAMS;
```

In this case, the `ioBase` element indicates the base I/O address range, the `irq` element is the number of the IRQ channel, and the `dma` element is the number of the DMA channel.

When the application in the configlette calls `usbTargTcdAttach()` to establish communication between the target application and the TCD, the HAL calls the single entry point of the TCD (see 6.3 Single Entry Point, p.142) with the function code `TCD_FNC_ATTACH` and passes in the pointer to this same set of TCD-defined parameters (see 6.5.1 Attaching the TCD, p.144).

### Detaching a TCD

The application in the configlette detaches the target application from a TCD by using `usbTargTcdDetach()` Pass this routine handle to the target channel that was obtained during the call to `usbTargTcdAttach()`.

The HAL responds to the `usbTargTcdDetach()` call by calling the single entry point of the TCD with the function code `TCD_FNC_DETACH` and the target channel handle (see 6.5.2 Detaching the TCD, p.146).

Figure 5-2 illustrates how the target application attaches to and detaches from a TCD.
The target layer maintains a callback table that lists the key entry points in the target application. Once the TCD and its associated target application have been attached to the target layer, the target layer can make asynchronous calls back into the target application by way of these callback routines.

The callback Table 5-1 below shows how the callback routines are mapped.

<table>
<thead>
<tr>
<th>Function</th>
<th>Callback Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>usbTargInitialize()</td>
<td>TCD_FNC_ATTACH</td>
</tr>
<tr>
<td>usbTargTcdAttach()</td>
<td>TARG_MNGMT_ATTACH</td>
</tr>
<tr>
<td>target channel handle</td>
<td></td>
</tr>
<tr>
<td>usbTargTcdDetach()</td>
<td>TCD_FNC_DETACH</td>
</tr>
<tr>
<td>usbTargShutdown()</td>
<td></td>
</tr>
<tr>
<td>mngmtFunc()</td>
<td>USB_TARG_MANAGEMENT_FUNC</td>
</tr>
<tr>
<td>featureClear()</td>
<td>USB_TARG_FEATURE_CLEAR_FUNC</td>
</tr>
<tr>
<td>featureSet()</td>
<td>USB_TARG_FEATURE_SET_FUNC</td>
</tr>
<tr>
<td>configurationGet()</td>
<td>USB_TARG_CONFIGURATION_GET_FUNC</td>
</tr>
</tbody>
</table>
Before the application in the configlette calls the `usbTargTcdAttach()` routine, the target application should set the function pointers in this table to point to the corresponding entry points that the target application implements. At any time after the application in the configlette calls `usbTargTcdAttach()`, the target layer may begin to make asynchronous calls to the entry points in the callback table. (Some callbacks occur during the attach process itself; others happen in response to activity on the USB.)

The target application should provide pointers in this table for each function that corresponds to a request that it intends to process, and `NULL` pointers for the others. The target layer provides default handling for any functions whose corresponding entry in this callback table is `NULL`.

**Example 5-2  Mass Storage Initializing and Attaching to a TCD**

The following code fragment demonstrates how the mass storage target application initializes and attaches with the TCD:

```c
#include "usbTargMs.h"

USB_TCD_NET2280_PARAMS paramsNET2280;

pUSB_TARG_CALLBACK_TABLE cbTable = &callbackTable;

usbTargMsCallbackInfo(&cbTable, &callbackParam);

/* Initialize usbTargLib */
```

**Table 5-1  Target Application Callback Table**

<table>
<thead>
<tr>
<th>Function</th>
<th>Callback Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>configurationSet()</code></td>
<td><code>USB_TARG_CONFIGURATION_SET_FUNC</code></td>
</tr>
<tr>
<td><code>descriptorGet()</code></td>
<td><code>USB_TARG_DESCRIPTOR_GET_FUNC</code></td>
</tr>
<tr>
<td><code>descriptorSet()</code></td>
<td><code>USB_TARG_DESCRIPTOR_SET_FUNC</code></td>
</tr>
<tr>
<td><code>interfaceGet()</code></td>
<td><code>USB_TARG_INTERFACE_GET_FUNC</code></td>
</tr>
<tr>
<td><code>interfaceSet()</code></td>
<td><code>USB_TARG_INTERFACE_SET_FUNC</code></td>
</tr>
<tr>
<td><code>statusGet()</code></td>
<td><code>USB_TARG_STATUS_GET_FUNC</code></td>
</tr>
<tr>
<td><code>addressSet()</code></td>
<td><code>USB_TARG_ADDRESS_SET_FUNC</code></td>
</tr>
<tr>
<td><code>synchFrameGet()</code></td>
<td><code>USB_TARG_SYNCH_FRAME_GET_FUNC</code></td>
</tr>
<tr>
<td><code>vendorSpecific()</code></td>
<td><code>USB_TARG_VENDOR_SPECIFIC_FUNC</code></td>
</tr>
</tbody>
</table>
if (usbTargInitialize () != OK)
{
    fprintf (fout,"usbTargInitialize() returned ERROR\n");
    return (ERROR);
}

/* This routine is defined in
* /target/config/comps/src/usrUsbTargInit.c.
* This routine obtains the TCD-defined structure value.
*/
sys2NET2280PciInit ();

paramsNET2280.ioBase [0] = BADDR_NET2280 [0];
paramsNET2280.irq = (UINT16) IRQ_NET2280;

/* Attach the NetChip NET2280 TCD to usbTargLib */

if (usbTargTcdAttach (usbTcdNET2280Exec, (pVOID) &paramsNET2280,
    callbackTable, callbackParam, &msTargChannel) != OK)
{
    fprintf (fout, "usbTargTcdAttach() returned ERROR\n");
    msTargChannel = NULL;
    return ERROR;
}

5.4 Enabling and Disabling the TCD

The target application may not yet be ready to begin handling USB requests when
the application in the configlette calls usbTargTcdAttach(), so the target layer
provides two additional routines to enable and to disable the TCD. Figure 5-3 is a
sequence diagram that shows how a TCD is enabled and disabled.

The application in the configlette uses usbTargEnable() to enable the specified
TCD. The TCD, in turn, enables the underlying target controller. Until the
application in the configlette calls usbTargEnable(), the target controller (and
thus, the peripheral) is not visible to the USB Host. In response to
usbTargEnable(), the HAL calls the single entry point of the TCD with the code
TCD_FNC_ENABLE.
The application in the configlette calls `usbTargDisable()` to disable the specified TCD typically just before detaching a TCD. In response to the `usbTargDisable()`, the HAL calls the single entry point of the TCD with the code `TCD_FNC_DISABLE`.

In both the `usbTargEnable()` and `usbTargDisable()` routines, the application in the configlette passes the channel handle that the application in the configlette received when it called `usbTargTcdAttach()` (see 5.3 Attaching and Detaching a TCD, p.105).
5.5 Implementing Target Application Callback Routines

The prototypes for each of the callbacks in the target application callback table are defined in `usbTargLib.h`. The individual callback routines and their additional parameters are explained in the following sections.

Callback and Target Channel Parameters

Two parameters are common to each of the callback routines:

- **param**
  - a parameter that is defined by the target application and that the application in the configlette passed to `usbTargTcdAttach()` as the **callbackParam**

- **targChannel**
  - the handle that the target layer assigned to the channel during the attach and returned from `usbTargTcdAttach()` in `pTargChannel` (this parameter is of the type `USB_TARG_CHANNEL`)

Control Pipe Request Callbacks

After the host issues a bus reset, it issues control requests to the device through the control endpoint. The target layer handles these requests and in turn calls the routines defined in the target application callback table (see Table 5-1).

The callback routines that handle these requests are listed in **Table 5-2**.

<table>
<thead>
<tr>
<th>Control Request</th>
<th>Callback Routine</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR_FEATURE</td>
<td>featureClear()</td>
<td>116</td>
</tr>
<tr>
<td>SET_FEATURE</td>
<td>featureSet()</td>
<td>116</td>
</tr>
<tr>
<td>GET_CONFIGURATION</td>
<td>configurationGet()</td>
<td>119</td>
</tr>
<tr>
<td>SET_CONFIGURATION</td>
<td>configurationSet()</td>
<td>119</td>
</tr>
<tr>
<td>GET_DESCRIPTOR</td>
<td>descriptorGet()</td>
<td>120</td>
</tr>
<tr>
<td>SET_DESCRIPTOR</td>
<td>descriptorSet()</td>
<td>120</td>
</tr>
</tbody>
</table>
5.5 Implementing Target Application Callback Routines

5.5.1 mngmtFunc() Callback

The target layer calls the mngmtFunc() callback in the target application callback table to inform the target application of changes that broadly affect USB operation, which it indicates by using certain management event codes. The target application may process or ignore these management event codes as it sees fit. This callback is defined as follows:

```c
STATUS mngmtFunc
(
    pVOID param, /* callback parameter*/
    USB_TARG_CHANNEL targChannel, /* target channel */
    UINT16 mngmtCode, /* management code */
    pVOID pContext, /* TCD specific parameter */
);
```

Management Code Parameter

The mngmtCode parameter tells the target application what management event has happened. The following valid management event codes are listed in usbTargLib.h:

- TARG_MNGMT_ATTACH – initial TCD attachment
- TARG_MNGMT_DETACH – TCD detach
- TARG_MNGMT_BUS_RESET – bus reset
- TARG_MNGMT_SUSPEND – suspend signal detected
- TARG_MNGMT_RESUME – resume signal detected
- TARG_MNGMT_DISCONNECT – disconnect signal detected
Context Value Parameter

Along with the management code, the target layer passes TCD-specific information to the target application in the `pContext` parameter. This information comes in the form of a `USB_APPLN_DEVICE_INFO` structure, shown below, which is populated by the TCD during the device attachment process.

```c
typedef struct usb_appln_device_info
{
    UINT32 uDeviceFeature;
    UINT32 uEndpointNumberBitmap
}USB_APPLN_DEVICE_INFO, *pUSB_APPLN_DEVICE_INFO;
```

In this structure, the `uDeviceFeature` is a bitmap that explains whether the device is USB 2.0 compliant, whether it supports Remote Wake Up, and whether it supports Test Mode. The TCD may set this bitmap to a combination of the bits `USB_FEATURE_DEVICE_REMOTE_WAKEUP`, `USB_FEATURE_TEST_MODE`, and `USB_FEATURE_USB20`.

These macros are defined in the file `/target/h/usb/target/usbHalCommon.h` and the application uses these macros only to interpret the structure member `uDeviceFeature` of the structure `USB_APPLN_DEVICE_INFO`. For example:

```c
üDeviceFeature = (USB_FEATURE_DEVICE_REMOTE | USB_FEATURE_USB20)
```

**NOTE:** You can also use bits 3 and 4 to specify “bus powered” and “self powered,” respectively.

The TCD sets `uEndpointNumberBitmap` to indicate the endpoints supported by the hardware. The first 16 bits indicate the OUT endpoint and the next 16 bits indicate the IN endpoint, with bits 0 to 15 corresponding to OUT endpoints 0 to 15, and bits 15 to 31 corresponding to IN endpoints 0 to 15.

Management Event Codes

These management event codes are defined as follows:

**TARG_MNGMT_ATTACH**

The target layer calls the `mgmtFunc()` routine in the target application with the `TARG_MNGMT_ATTACH` code during the attachment process. The `pContext` parameter is set according to the description in *Context Value Parameter*, p.114.
TARG_MNGMT_DETACH
The target layer calls the management routine in the target application with this code during the detachment process. The pContext parameter is NULL in this case. During this detachment process the target application resets the device descriptor values to their initial values.

TARG_MNGMT_BUS_RESET
The target layer calls the management routine in the target application with this code when the host issues a bus reset event.

During a bus reset event, the TCD provides the target application with the speed at which the target controller is operating (by passing either USB_TCD_FULL_SPEED, USB_TCD_LOW_SPEED, or USB_TCD_HIGH_SPEED in the pContext parameter (see Context Value Parameter, p.114) to mngmtFunc()).

Depending on the operating speed of the device the target application should modify various descriptors value as specified in the USB 2.0 Specification.

Example 5-3  Bus Reset Example

The following code fragment illustrates the bus reset event:

```c
LOCAL STATUS mngmtFunc(
    pVOID param, /* callback parameter */
    USB_TARG_CHANNEL targChannel, /* target channel */
    UINT16 mngmtCode, /* management code */
    pVOID pContext /* TCD specific context value */
) {

    /* USB_APPLN_DEVICE INFO */
    pUSB_APPLN_DEVICE_INFO pDeviceInfo = NULL;

    switch (mngmtCode) {
    case TARG_MNGMT_ATTACH:
        ...
        break;

    case TARG_MNGMT_DETACH:
        ...
        break;

    case TARG_MNGMT_BUS_RESET:
        if (g_uSpeed == USB_TCD_HIGH_SPEED)
            {
                /* update the maxpacket size field in device descriptor *
                 * and device qualifier depending on the speed */
```
... /* update the max packet size for generic endpoints */...
break;
}
}

TARG_MNGMT_DISCONNECT
When the target layer calls the management routine in the target application with this code, the target application releases its endpoints. The pContext parameter is NULL in this case.

TARG_MNGMT_SUSPEND and TARG_MNGMT_RESUME
The target layer calls the management routine in the target application with these codes when the TCD notifies the target layer about suspend and resume events. The pContext parameter is NULL in these cases.

5.5.2 Clear and Set Callbacks

The target layer calls the target application’s featureClear( ) and featureSet( ) callbacks in response to CLEAR_FEATURE and SET_FEATURE requests from the host.

The featureClear() routine should return ERROR if the device is in the default state (in other words, if the device address is zero), if the feature does not exist or cannot be cleared (for instance, because it is the “test mode” feature or an unsupported feature), or if the endpoint does not exist. The featureClear() callback is defined as follows:

```c
STATUS featureClear
(
    pVOID param,
    USB_TARG_CHANNEL targChannel,
    UINT8 requestType,
    UINT16 feature,
    UINT16 index
);
```

The featureSet() routine should return ERROR if the device is in the default state and the feature to be set is anything other than the “test mode” feature, if the feature cannot be set for any other reason, or if the endpoint or interface does not exist. The featureSet() callback is defined as follows:

```c
STATUS featureSet
(
    pVOID param,
    USB_TARG_CHANNEL targChannel,
    UINT8 requestType,
```
USB Peripheral Stack Target Layer Overview

5.5 Implementing Target Application Callback Routines

```c
UINT16 feature,
UINT16 index
);
```

Request Type Parameter

The `requestType` parameter must be set to either `USB_REQ_CLEAR_FEATURE` or `USB_REQ_SET_FEATURE`. See Table 5-3 for the list of standard `requestType` values.

<table>
<thead>
<tr>
<th>Request Type</th>
<th>numerical value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_REQ_GET_STATUS</td>
<td>0</td>
</tr>
<tr>
<td>USB_REQ_CLEAR_FEATURE</td>
<td>1</td>
</tr>
<tr>
<td>USB_REQ_GET_STATE</td>
<td>2</td>
</tr>
<tr>
<td>USB_REQ_SET_FEATURE</td>
<td>3</td>
</tr>
<tr>
<td>USB_REQ_SET_ADDRESS</td>
<td>5</td>
</tr>
<tr>
<td>USB_REQ_GET_DESCRIPTOR</td>
<td>6</td>
</tr>
<tr>
<td>USB_REQ_SET_DESCRIPTOR</td>
<td>7</td>
</tr>
<tr>
<td>USB_REQ_GET_CONFIGURATION</td>
<td>8</td>
</tr>
<tr>
<td>USB_REQ_SET_CONFIGURATION</td>
<td>9</td>
</tr>
<tr>
<td>USB_REQ_GET_INTERFACE</td>
<td>10</td>
</tr>
<tr>
<td>USB_REQ_SET_INTERFACE</td>
<td>11</td>
</tr>
<tr>
<td>USB_REQ_SYNCH_FRAME</td>
<td>12</td>
</tr>
</tbody>
</table>

The target application should call the appropriate interface exposed by the target layer, depending upon whether the request is to set or clear.

**USB_REQ_CLEAR_FEATURE Request**

If the host requests to clear the halt condition on the endpoint, the target application should call the `usbTargPipeStatusSet()` routine.

```c
if (usbTargPipeStatusSet (handle, TCD_ENDPOINT_UNSTALL) == ERROR)
{
```
/* Unable to unstall the endpoint */
return ERROR
}

If the host requests to clear a device specific feature, the target application should call `usbTargDeviceFeatureClear()` routine.

if (usbTargDeviceFeatureClear (handle, feature) == ERROR)
{
    /* Unable to clear the device feature*/
    return ERROR
}

USB_REQ_SET_FEATURE Request

If the host requests to set the halt condition on the endpoint, the target application should call `usbTargPipeStatusSet()` routine.

if (usbTargPipeStatusSet (handle, TCD_ENDPOINT_STALL) == ERROR)
{
    /* Unable to unstall the endpoint */
    return ERROR
}

If the host requests to set a device specific feature, the target application should call `usbTargDeviceFeatureSet()` routine.

if (usbTargDeviceFeatureSet(handle, feature, testSelector) == ERROR)
{
    /* Unable to clear the device feature*/
    return ERROR
}

Feature Parameter

If the target application supports these functions, it should do so by clearing or setting the feature identified by the `feature` and `index` parameters. Table 5-4 lists the valid `feature` values.

<table>
<thead>
<tr>
<th>Feature Selector</th>
<th>Recipient</th>
<th>Feature Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_FSEL_DEV_ENDPOINT_HALT</td>
<td>Endpoint</td>
<td>0</td>
</tr>
<tr>
<td>USB_FSEL_DEV_REMOTE_WAKEUP</td>
<td>Device</td>
<td>1</td>
</tr>
<tr>
<td>USB_FSEL_DEV_TEST_MODE</td>
<td>Device</td>
<td>2</td>
</tr>
</tbody>
</table>
Index Parameter

For the index parameter, if the recipient of the request (as shown in Table 5-4) is a device, \textit{index} is zero; if the recipient is an endpoint, then \textit{index} is the endpoint address of that endpoint.

5.5.3 \texttt{configurationGet()} Callback

The target layer calls the \texttt{configurationGet()} target application callback in response to a \texttt{GET\_CONFIGURATION} request from the host. The \texttt{configurationGet()} callback is defined as follows:

\begin{verbatim}
STATUS configurationGet
(   pVOID param,
   USB_TARG_CHANNEL targChannel,
   pUINT8 pConfiguration
);
\end{verbatim}

In the \texttt{configurationGet()} callback, the target application should fill the \texttt{pConfiguration} buffer with a \texttt{UINT8} value that matches with the configuration value in the current configuration descriptor.

On successful return from \texttt{configurationGet()}, the target layer transmits this configuration information back to the USB host. If the device is in the default state (that is, if the device address is zero), the target application should return \texttt{ERROR} from \texttt{configurationGet()}.

5.5.4 \texttt{configurationSet()} Callback

The target layer calls the \texttt{configurationSet()} target application callback in response to a \texttt{SET\_CONFIGURATION} request from the host. The \texttt{configurationSet()} callback is defined as follows:

\begin{verbatim}
STATUS configurationSet
(   pVOID param,
   USB_TARG_CHANNEL targChannel,
   UINT8 configuration
);
\end{verbatim}

In the \texttt{configurationSet()} callback, the target application should set the current configuration to the one indicated by the \texttt{configuration} value (assuming the value matches with a valid configuration descriptor). If the configuration value is
invalid, or if the device is in the default state, the target application should return ERROR from configurationSet().

5.5.5 descriptorGet() and descriptorSet() Callbacks

The target layer calls the descriptorGet() and descriptorSet() target application callbacks in response to GET_DESCRIPTOR and SET_DESCRIPTOR requests from the host. The requestType, descriptorType, descriptorIndex, and languageId parameters identify the descriptor. The descriptorGet() callback is defined as follows:

```c
STATUS descriptorGet
{
    pVOID        param,
    USB_TARG_CHANNEL targChannel,
    UINT8        requestType,
    UINT8        descriptorType,
    UINT8        descriptorIndex,
    UINT16       languageId,
    UINT16       length,
    pUINT8       pBfr,
    pUINT16      pActLen
};
```

The descriptorSet() callback is defined as follows:

```c
STATUS descriptorSet
{
    pVOID        param,
    USB_TARG_CHANNEL targChannel,
    UINT8        requestType,
    UINT8        descriptorType,
    UINT8        descriptorIndex,
    UINT16       languageId,
    UINT16       length,
    pUINT8       pBfr,
    pUINT16      pActLen
};
```

**Request Type Parameter**

Table 5-5 lists the constants associated with requestType bits.

Bit 7 of requestType specifies the direction, so requestType must be masked with USB_RT_HOST_TO_DEV or USB_RT_DEV_TO_HOST to specify the direction.
Bits 5 and 6 of `requestType` specify the request type, so `requestType` must be masked with `USB_RT_STANDARD`, `USB_RT_CLASS`, or `USB_RT_VENDOR` to specify the request type.

Bits 0 through 4 of `requestType` specify the recipient, so `requestType` must be masked with `USB_RT_DEVICE`, `USB_RT_INTERFACE`, `USB_RT_ENDPOINT`, or `USB_RT_OTHER` to indicate the recipient.

**Descriptor Type and Index Parameters**

Table 5-6 lists the valid `descriptorType` values (the `descriptorIndex` specifies a specific descriptor when multiple descriptors of the same type are implemented on a device).
Language ID Parameter

The `languageId` parameter is used for the string descriptor; the possible two-byte `languageId` values can be found at http://www.usb.org/developers/docs/USB_LANGIDs.pdf.

Length and Buffer Parameters

In the `descriptorGet()` callback, the target application should fill `pBfr` with the requested descriptor (truncated if it is larger than the buffer size, which is specified by `length`). The target application should also set `pActLen` to the actual length of the descriptor placed in `pBfr`. The target layer transmits the descriptor back to the USB host. If the target application does not support the requested descriptor, it should return `ERROR` from this callback, which will trigger a Request Error.

If the target application supports the `SET_DESCRIPTOR` request, it should set descriptors with the specified values in the `descriptorSet()` callback routine. The `length` parameter indicates the length of the descriptor that will be sent by the host in the data stage, as specified in the setup packet.

The `descriptorSet()` callback should set `pActLen` to the actual length of the descriptor that it set. If the two values, `length` and `pActLen`, are not identical,
5 USB Peripheral Stack Target Layer Overview

5.5 Implementing Target Application Callback Routines

5.5.6 interfaceGet( ) Callback

The target layer calls the `interfaceGet()` target application callback in response to a GET_INTERFACE request from the host. The `interfaceGet()` callback is defined as follows:

```c
STATUS interfaceGet
  (  
    pVOID param,
    USB_TARG_CHANNEL targChannel,  
    UINT16 interface,
    pUINT8 pAlternateSetting  
  );
```

In the `interfaceGet()` callback, the target application should store the alternate interface setting of the interface specified by `interface` in the variable `pAlternateSetting`. The target application should return `ERROR` from `interfaceGet()` if the device is in the default or the addressed state.

5.5.7 interfaceSet( ) Callback

The target layer calls the `interfaceSet()` target application callback in response to a SET_INTERFACE request from the host. The `interfaceSet()` callback is defined as follows:

```c
STATUS interfaceSet
  (  
    pVOID param,
    USB_TARG_CHANNEL targChannel,  
    UINT16 interface,
    UINT8 alternateSetting  
  );
```

In the `interfaceSet()` callback the target application should set the alternate setting of the interface specified by `interface` to the `alternateSetting` setting. The target application should return `ERROR` from `interfaceSet()` if the device is in the default or the addressed state or if the alternate setting is invalid.

descriptorSet() should return `ERROR`, which will cause the target layer to stall the control endpoints.
5.5.8 statusGet( ) Callback

The target layer calls the `statusGet()` target application callback in response to a `GET_STATUS` request from the host. The `statusGet()` callback is defined as follows:

```c
STATUS statusGet
{
    pVOID param,
    USB_TARG_CHANNEL targChannel,
    UINT16 requestType,
    UINT16 index,
    UINT16 length,
    pUINT8 pBfr
};
```

If the target application supports this callback, it should do so by storing the status of the recipient identified by `requestType` in `pBfr`. If the recipient is an endpoint, `index` specifies the endpoint address of the endpoint. The `length` parameter is set to the same value as the `wLength` value of the setup packet, that is: 2. For appropriate status values and their meanings, see Table 5-7.

<table>
<thead>
<tr>
<th>Recipient Status</th>
<th>Associated Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipient is a device that is not self-powered and is not enabled to request remote wakeup</td>
<td>0</td>
</tr>
<tr>
<td>Recipient is a device that is self-powered, but is not enabled to request remote wakeup</td>
<td>USB_DEV_STS_LOCAL_POWER</td>
</tr>
<tr>
<td>Recipient is a device that is not self powered, but is enabled to request remote wakeup</td>
<td>USB_DEV_STS_REMOTE_WAKEUP</td>
</tr>
<tr>
<td>Recipient is a device that is self-powered and enabled to request remote wakeup</td>
<td>USB_DEV_STS_LOCAL_POWER</td>
</tr>
<tr>
<td>Recipient is an interface 0</td>
<td>0</td>
</tr>
</tbody>
</table>
5.5 Implementing Target Application Callback Routines

5.5.9 addressSet() Callback

The target layer calls the addressSet() callback in response to a SET_ADDRESS request from the host. The addressSet() callback is defined as follows:

```c
STATUS addressSet
{
pVOID param,
USB_TARG_CHANNEL targChannel,
UINT16 deviceAddress
}
```

The target layer passes a deviceAddress parameter between 0 and 127, inclusive. The target application can use this device address as an identifier for the device in the bus. The target application should return ERROR from its addressSet() callback if the device has passed the configuration stage.

5.5.10 synchFrameGet() Callback

It is sometimes necessary for clients to re-synchronize with devices when the two are exchanging data isochronously. This function allows a client to query a reference frame number maintained by the device (indicated by pFrameNo). Please refer to the USB 2.0 specification for more detail.

The target layer calls the synchFrameGet() callback in response to a SYNCH_FRAME request from the host. The synchFrameGet() callback is defined as follows:

```c
STATUS synchFrameGet
{
pVOID param,
```
USB_TARG_CHANNEL targChannel,
UINT16 endpoint,
pUINT16 pFrameNo /* Frame Number */
);

If the specified endpoint does not support the SYNCH_FRAME request, this routine should return ERROR, which will trigger a Request Error.

### 5.5.11 vendorSpecific( ) Callback

The target layer routes all requests that it receives on a peripheral’s default control pipe, which are not recognized as standard requests to the target application’s vendorSpecific( ) callback, if the target application provides such a callback. The vendorSpecific( ) callback is defined as follows:

```c
STATUS vendorSpecific( 
    pVOID  param,
    USB_TARG_CHANNEL targChannel,
    UINT8 requestType,
    UINT8 request,
    UINT16 value,
    UINT16 index,
    UINT16 length
);
```

The `value`, `index`, `length`, `requestType`, and `request` parameters correspond to the `wValue`, `wIndex`, `wLength`, `bmRequestType`, and `bRequest` fields in the USB device request setup packet that the host expects to follow this vendor-specific call. Target applications handle vendor-specific requests in various ways. Some vendor-specific requests may require additional data transfer, perhaps using target layer routines like `usbTargControlPayloadRcv( )` or `usbTargControlResponseSend( )`. If there is no data transfer associated with a vendor specific request, use `usbTargControlStatusSend( )` instead. (See 5.6.4 Handling Default Pipe Requests, p.136).

### 5.6 Pipe-Specific Requests

The TARG_PIPE data structure holds information about a pipe. This structure is defined in `usbTargLib.h` as follows:

```c
typedef struct targPipe  /* TARG_PIPE */
```
The members of this structure hold the following information:

**pipeHandle**

The pipe handle that the target application uses when carrying out USB transfer on this endpoint. This is just an identifier for the pipe, and does not represent the endpoint number.

**pHalPipeHandle**

A pointer to HAL information that is used for internal bookkeeping within the target layer. Whenever the target application creates a pipe, a pointer to structure USB_HAL_PIPE_INFO is stored in this handle. This abstracts all the endpoint specific information form the TargLib layer and the TargLib layer uses this handle to communicate with the particular endpoint.

Note that the targLib layer need not know the integrate details of the endpoint.

**pTargTcd**

This is the pointer to a structure that the target layer uses to maintain information about Target Controller Drivers. Your target application does not need to refer to this data.

### 5.6.1 Creating and Destroying the Pipes

A target application uses the target layer routines `usbTargPipeCreate()` and `usbTargPipeDestroy()` to create and destroy pipes.

The `usbTargPipeCreate()` routine creates a pipe for communication on a specific target endpoint and returns a pipe handle (in `pPipeHandle`). The target application uses this pipe handle to communicate with the endpoint.

The target application sets the endpoint descriptor (for details about this structure, see *Endpoint Descriptor*, p.128), the configuration value of the device, the number of the interface that supports the endpoint, and the alternate setting of the interface. In response to the `usbTargPipeCreate()`, the HAL calls the single entry point of the TCD and passes it a TRB that contains the code TCD_FNC_ENDPOINT_ASSIGN and this same endpoint descriptor, configuration value, interface number, and alternate setting (for details about this TRB, see *6.5.8 Assigning the Endpoints*, p.148).
The default control pipes (endpoint #0) are created and maintained by the target layer itself when the host sends a bus reset event. The pipe handle to the control endpoints is not exposed to the target application. If the target application needs to carry out any communication with the host using the default control pipes it can do so with the `usbTargControlResponseSend()`, `usbTargControlStatusSend()`, and `usbTargControlPayloadRcv()` routines (see 5.6.4 Handling Default Pipe Requests, p.136).

### Endpoint Descriptor

The endpoint descriptor is a `USB_ENDPOINT_DESCR` structure containing the elements described below.

- **length**
  - a UINT8 containing the total length in bytes of the endpoint descriptor

- **descriptorType**
  - a UINT8 containing the endpoint descriptor type: `USB_DESCR_ENDPOINT`

- **endpointAddress**
  - a UINT8 containing the address of the endpoint, encoded as follows:
    - bits 3 to 0 – the endpoint number
    - bits 6 to 4 – reserved
    - bit 7 – the direction (ignored for control endpoints); 0=OUT, 1=IN

- **attributes**
  - a UINT8 containing the endpoint attributes when it is configured using the `bConfigurationValue`:
    - bits 1 to 0 – transfer type
      - 00 – control
      - 01 – isochronous
      - 10 – bulk
      - 11 – interrupt
    - bits 3 to 2 – synchronization type
      - 00 – no synchronization
      - 01 – asynchronous
      - 10 – adaptive
      - 11 – synchronous

If not an isochronous endpoint, bits 5 to 2 are reserved and must be set to zero; if isochronous, they are defined as follows:
- bits 5 to 4 – usage type
  - 00 – data endpoint
  - 01 – feedback endpoint
  - 10 – implicit feedback data endpoint
  - 11 – reserved
  - all other bits are reserved and must be reset to zero

$maxPacketSize$

a UINT16 containing the maximum packet size that this endpoint is capable of sending or receiving in this configuration.

For isochronous endpoints, this value reserves the bus time in the schedule, which is required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that which is reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB-defined mechanisms.

For all endpoints, bits 10 through 0 specify the maximum packet size in bytes.

For high-speed isochronous and interrupt endpoints, bits 12 and 11 indicate the number of additional transaction opportunities per microframe:

- 00 – 1 additional (2 per microframe)
- 10 – 2 additional (3 per microframe)
- 11 – reserved
- bits 15 through 13 are reserved and must be set to zero

$interval$

a UINT8 containing an interval for polling the endpoint for data transfers (expressed in frames or in microframes depending on the device operating speed – in other words, in 1 millisecond or 125 microsecond units)

- For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16.
- For full-/low-speed interrupt endpoints, the value may range from 1 to 255.
- For high-speed interrupt endpoints, the value must be in the range from 1 to 16.
- For high-speed bulk/control OUT endpoints, the value must specify the maximum NAK rate of the endpoint. A value of 0 indicates that the endpoint never NAKs. Other values indicate at most 1 NAK per $interval$ number of microframes. This value may range from 0 to 255.
The target application tears down a previously created pipe by calling `usbTargPipeCreate()`. When the target application calls this routine, the HAL releases the endpoint for the pipe indicated by `pipeHandle`, releases that pipe handle, and calls the single entry point of the TCD with the function code `TCD_FNC_ENDPOINT_RELEASE`.

### 5.6.2 Transferring and Aborting Data

Data transfer can be of two types: generic and control:

**generic data transfer**
- The target application uses the target layer routine `usbTargTransfer()` to do generic data transfer between the peripheral stack and the USB host, see Figure 5-4.

**control data transfer**
- If the target application instead wants to communicate with the host using the default control pipes it can do so by using the `usbTargControlResponseSend()`, `usbTargControlStatusSend()`, and `usbTargControlPayloadRcv()` routines (see 5.6.4 Handling Default Pipe Requests, p.136).

The `usbTargTransfer()` routine initiates a transfer on the pipe indicated by `pipeHandle`. The data to be transferred is described by a structure, `USB_ERP`, that the target application must allocate and initialize before it calls `usbdTargTransfer()`.
USB_ERP Structure

The structure `USB_ERP` is defined in `usb.h` and takes the following form:

```c
typedef struct usb_erp
{
    LINK targLink;        /* link field used internally by usbTargLib */
    pVOID targPtr;        /* ptr field for use by usbTargLib */
    LINK tcdLink;         /* link field used internally by USB TCD */
    pVOID tcdPtr;         /* ptr field for use by USB TCD */
    pVOID userPtr;        /* ptr field for use by client */
    UINT16 erpLen;        /* total length of ERP structure */
} usb_erp;
```
The elements of the `USB_ERP` structure are listed below with their descriptions.

### targLink
This is not used by either the target layer or the HAL, and may be used by the target application. The target application can use this link structure to maintain a list of ERPs by casting the `USB_ERP` structure as a `LINK` element of a linked list.

### targPtr
The target layer uses this pointer in the case of control transfers to point to a `TARG_TCD` structure. This element is not used for generic endpoint transfers.

### tcdLink
The HAL uses this element to maintain a linked list of the ERPs that have been submitted to the TCD.

### tcdPtr
The HAL uses this pointer to the TCD when it wants to send a zero-length packet.

### userPtr
The target application may use this pointer for its own purposes.

### erpLen
The target application sets this element to the total size of the `USB_ERP` structure including the `bfrList` array – that is, to:

\[
\text{sizeof(USB_ERP) + (sizeof(USB_BFR_DESCR) * (bfrCount - 1))}
\]

### result
The target layer sets this element to the ERP completion result, either `OK` or one of the `S_usbTcdLib_xxxx` codes listed below.

- `OK`
- `S_usbTcdLib_BAD_PARAM`
- `S_usbTcdLib_BAD_HANDLE`
- `S_usbTcdLib_OUT_OF_MEMORY`
5 USB Peripheral Stack Target Layer Overview

5.6 Pipe-Specific Requests

S_usbTcdLib_OUT_OF_RESOURCES
S_usbTcdLib_NOT_IMPLEMENTED
S_usbTcdLib_GENERAL_FAULT
S_usbTcdLib_NOT_INITIALIZED
S_usbTcdLib_INT_HOOK_FAILED
S_usbTcdLib_HW_NOT_READY
S_usbTcdLib_NOT_SUPPORTED
S_usbTcdLib_ERP_CANCELED
S_usbTcdLib_CANNOT_CANCEL
S_usbTcdLib_SHUTDOWN
S_usbTcdLib_DATA_TOGGLE_FAULT
S_usbTcdLib_PID_MISMATCH
S_usbTcdLib_COMM_FAULT
S_usbTcdLibSTALL_ERROR
S_usbTcdLib_NEW_SETUP_PACKET
S_usbTcdLib_DATA_OVERRUN

**NOTE:** Note that the target application should not consider this field to be accurately set until the target layer invokes the `userCallback()` routine.

`targCallback()`

The target layer sets this element which it uses internally.

`userCallback()`

The target application sets this element to a routine that the target layer calls when the ERP has either been successfully transmitted or there has been an error. The target application should check the `result` field of the ERP in this callback routine and respond appropriately. An `ERP_CALLBACK` has the following form:

void myErpCallback( pVOID pErp );

`pPipeHandle`

The HAL and target layer use this pipe handle internally. The target application does not need to set it or use it.
**transferType**  
The target layer sets this element based on the ERP type:  
`USB_XFRTYPE_CONTROL`, `USB_XFRTYPE_ISOCH`,  
`USB_XFRTYPE_INTERRUPT`, or `USB_XFRTYPE_BULK`.

**dataToggle**  
The target layer sets this element either to `USB_DATA0` or `USB_DATA1`.

**bfrCount**  
The target application sets this to the number of buffers in `bfrList`.

**bfrList**  
The target application sets this to a `USB_BFR_LIST` structure or to an array of  
`USB_BFR_LIST` structures. This structure describes a block of data in the  
transfer. See Table 5-8.

**endpointId**  
The target layer sets this element based on the value recorded when the pipe  
was created.

---

**Table 5-8 Elements of the USB_BFR_LIST Structure**

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| **pid** | specifies the packet type: `USB_PID_SETUP`, `USB_PID_IN` or `USB_PID_OUT`;  
          if the first `USB_BFR_LIST` structure in the `bfrList` array has  
          a pid of `USB_PID_SETUP` then it must be the only such structure  
          in the `bfrList` array; a `bfrList` array must not contain  
          structures that have both `USB_PID_IN` and `USB_PID_OUT` pid  
          values, but they must only have one or the other. `USB_PID_IN`  
          indicates that the packet is going from the target to the host;  
          `USB_PID_OUT` indicates that the packet is going from the host to  
          the target.                                                                 |
| **pBfr** | the contents of the buffer                                              |
| **bfrLen** | the size of the buffer                                              |
| **actLen** | the HAL sets this, on completion of the data transfer, to the actual  
             amount of this buffer that has been transmitted                 |

The target application should check this value in the  
`userCallback()` function.
usbTargTransfer( ) Routine

The first parameter to usbTargTransfer( ) is the handle to the pipe on which the data is to be transferred, and the second parameter is the ERP to be transferred. The HAL responds when this routine is called by calling the single entry point of the TCD with the code TCD_FNC_COPY_DATA_FROM_EPBUF or TCD_FNC_COPY_DATA_TO_EPBUF depending on the direction for which that pipe is supported.

The usbTargTransfer() routine simply enqueues the ERP in the list of ERP to be transferred on that endpoint. The result of the ERP is obtained by querying the result field of the ERP structure in the callback routine. For a successful transfer, the result field would be set to OK. The target application callback routine would be called by the HAL.

Aborting a Data Transfer

The target application can call the usbTargTransferAbort( ) routine to abort a transfer that it previously submitted with the usbTargTransfer( ) routine. When a transfer is aborted, the userCallback() referenced in pErp will be called. The result element of the ERP will be set to S_usbTcdLib_ERP_CANCELLED and the actLen element will indicate whether any of the data in the ERP was transferred.

5.6.3 Stalling and Un-stalling the Endpoint

The target application may stall or un-stall the generic endpoints. It should stall the endpoint in response to certain error conditions. The STALL state indicates to the host that an error has occurred on the target. A target application uses the target layer routine usbTargPipeStatusSet() to stall or un-stall a particular endpoint. This routine sets the state of a pipe to TCD_ENDPOINT_STALL or TCD_ENDPOINT_UNSTALL.

NOTE: Halt and stall are used interchangeably and have essentially the same meaning.

The HAL responds to this routine by calling the single entry point in the TCD with the code TCD_FNC_ENDPOINT_STATE_SET. The state is either TCD_ENDPOINT_STALL or TCD_ENDPOINT_UNSTALL.
The target application can use the `usbTargPipeStatusGet( )` target layer routine to determine whether the endpoint to a pipe is stalled or not. The HAL responds to this routine by calling the single entry point of the TCD with the code `TCD_FNC_ENDPOINT_STATUS_GET` and by storing the status of the endpoint in the `pBuf` parameter (either `USB_ENDPOINT_STS_HALT` if the endpoint is stalled, or zero otherwise).

### 5.6.4 Handling Default Pipe Requests

The target application uses these routines to transfer data on the default control pipe:

- `usbTargControlResponseSend( )`
- `usbTargControlStatusSend( )`
- `usbTargControlPayloadRecv( )`

The target application uses the `usbTargControlResponseSend( )` routine to send control pipe responses to the host using `usbTargControlResponseSend( )`. The target application uses this sent data on various Control–IN requests for the host. For example, on a `GET_DESCRIPTOR` request, the target application calls this API to send the data in response to the request from host.

The target application uses the `usbTargControlStatusSend( )` routine to send the status to the host when the control transfer does not have a data stage.

The target application calls the `usbTargControlPayloadRecv( )` routine to request that it receive control pipe responses from the host in a callback function `usbTargControlPayloadRecv( )`. After the target application calls this routine, when control data is received from the host, the `userCallback( )` routine will be invoked and at that time the `pBfr` will contain this control data.

**NOTE:** The target application cannot set the `TCD_ENDPOINT_STALL` or `TCD_ENDPOINT_UNSTALL` state directly for the default control pipe. The target layer sets the default control pipe to the STALL state when it detects an error in the processing of a request on the default control pipe, for instance when the target application returns `ERROR` from a callback designed to handle a standard request.
5.7 Device Control and Status Information

5.7.1 Getting the Frame Number

Some target applications, particularly those that implement isochronous data transfers, need to determine the current USB frame number. Use the target layer routine `usbTargCurrentFrameGet()` for this purpose.

The target layer stores the current frame number in the `pFrameNo` parameter. The HAL responds to a call to `usbTargCurrentFrameGet()` by calling the single entry point in the TCD with the code `TCD_FNC_CURRENT_FRAME_GET`.

5.7.2 Resuming the Signal

A target application can drive `RESUME` signaling, as defined in the USB 2.0 specification, by using the target layer routine `usbTargSignalResume()` if the USB device is in the `SUSPEND` state, a target application can use this routine to resume the device.

The HAL responds to this routine by calling the single entry point of the TCD with the code `TCD_FNC_SIGNAL_RESUME`.

5.7.3 Setting and Clearing a Device Feature

Many devices support the Remote Wakeup feature and the Test Mode feature. A target application can set these features and clear the Remote Wakeup feature with the target layer routines `usbTargDeviceFeatureSet()` and `usbTargDeviceFeatureClear()` in response to a `SET_FEATURE` or `CLEAR_FEATURE` request from the host.

The second parameter to `usbTargDeviceFeatureSet()` is `ufeatureSelector`, which specifies whether we want to set the Remote Wakeup feature or the Test Mode feature (see Table 5-4 for a list of feature selector values). The third parameter, `uTestSelector`, specifies the Test Selector values (see Table 5-9 for a list of test selector values).

The HAL responds to this routine by calling the single entry point of the TCD with the code `TCD_FNC_DEVICE_FEATURE_SET`. 
The HAL responds to this routine by calling the single entry point of the TCD with the code `TCD_FNC_DEVICE_FEATURE_CLEAR`.

### 5.8 Shutdown Procedures

The Target Layer exposes a routine `usbTargShutdown()` . This routine can be used by the target application or the configlette to shut down the Wind River USB peripheral stack.

The Target Layer maintains a usage count `initCount`, which get incremented every time `usbTargInitialize()` is called. Similarly, on every corresponding call to `usbTargShutDown()` , the usage count is decremented. Only when the usage count reaches 0 are all the attached TCDs detached, and all the resources allocated by the target layer released. **Figure 5-5** shows the target layer shutdown process.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB_TEST_MODE_J_STATE</td>
<td>1</td>
</tr>
<tr>
<td>USB_TEST_MODE_K_STATE</td>
<td>2</td>
</tr>
<tr>
<td>USB_TEST_MODE_SE0_ACK</td>
<td>3</td>
</tr>
<tr>
<td>USB_TEST_MODE_TEST_PACKET</td>
<td>4</td>
</tr>
<tr>
<td>USB_TEST_MODE_TEST_FORCE_ENABLE</td>
<td>5</td>
</tr>
</tbody>
</table>

**NOTE:** According to the USB 2.0 specification, the Test Mode feature cannot be cleared.
Figure 5-5  Target Layer Shutdown

```
Configlette  Target Layer  HAL  TCD

usbTargShutdown()

TCD_FNC_DETACH
```
6

Target Controller Drivers

6.1 Introduction 141
6.2 Hardware Adaptation Layer Overview 142
6.3 Single Entry Point 142
6.4 Target Request Block 142
6.5 Function Codes 144

6.1 Introduction

This chapter shows how to create a target controller driver that interfaces with the Hardware Adaptation Layer in the Wind River USB peripheral stack. For instructions on how to create a target application that interfaces with the target layer in the Wind River USB peripheral stack, see 5. USB Peripheral Stack Target Layer Overview.
6.2 Hardware Adaptation Layer Overview

The Hardware Adaptation Layer (HAL) sits between the target controller driver and the target layer in the peripheral stack. It carries out all of the hardware-independent functions of the target controller and makes the implementation of higher stack layers independent of the TCD.

This section explains how HAL calls the single entry point exposed by the TCD to carry out different operations on the target controller. The HAL calls the single entry point exposed by the TCD with specific function codes in a TRB (Target Request Block). The code describes the services requested from the TCD. 6.4 Target Request Block, p.142 explains the TRB and 6.5 Function Codes, p.144 explains the different function codes. All TCDs must implement the single entry point that provides service for these function codes. It is not necessary for the TCD to implement each function code, only those that it plans to respond to in a device-specific way.

6.3 Single Entry Point

The target controller driver exposes a single entry point to the HAL. The HAL makes all requests to the TCD by passing an appropriate TRB with a proper function code to this single entry point, which has the form:

```c
STATUS usbTcdXXXXExec( pVOID pTrb )
```

The `pTrb` parameter contains the TRB corresponding to the particular request.

6.4 Target Request Block

The HAL makes requests to the TCD by constructing a Target Request Block (TRB) and passing it to the single entry point of the TCD. TRBs begin with a common header that may be followed by parameters specific to the function being requested. This common TRB header is shown below:

```c
typedef struct trb_header
```
The handle element of the TRB header is the handle to the TCD. The TCD creates this handle and fills in the handle element when the TCD is called with the TCD_FNC_ATTACH code. The HAL then keeps track of this handle and uses it to fill in the handle element of TRBs for subsequent requests.

The HAL sets the function element in the TRB header to one of the TCD function requests shown in Table 6-1.

### Table 6-1: TCD Function Requests

<table>
<thead>
<tr>
<th>Function Code</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCD_FNC_ATTACH</td>
<td>144</td>
</tr>
<tr>
<td>TCD_FNC_DETACH</td>
<td>146</td>
</tr>
<tr>
<td>TCD_FNC_ENABLE</td>
<td>146</td>
</tr>
<tr>
<td>TCD_FNC_DISABLE</td>
<td>146</td>
</tr>
<tr>
<td>TCD_FNC_ADDRESS_SET</td>
<td>147</td>
</tr>
<tr>
<td>TCD_FNC_SIGNAL_RESUME</td>
<td>147</td>
</tr>
<tr>
<td>TCD_FNC_DEVICE_FEATURE_SET</td>
<td>147</td>
</tr>
<tr>
<td>TCD_FNC_DEVICE_FEATURE_CLEAR</td>
<td>147</td>
</tr>
<tr>
<td>TCD_FNC_CURRENT_FRAME_GET</td>
<td>148</td>
</tr>
<tr>
<td>TCD_FNC_ENDPOINT.Assign</td>
<td>148</td>
</tr>
<tr>
<td>TCD_FNC_ENDPOINT_RELEASE</td>
<td>149</td>
</tr>
<tr>
<td>TCD_FNC_ENDIANPSTSET</td>
<td>149</td>
</tr>
<tr>
<td>TCD_FNC_ENDPOINT_STATUS_GET</td>
<td>150</td>
</tr>
<tr>
<td>TCD_FNC_IS_BUFFER_EMPTY</td>
<td>151</td>
</tr>
<tr>
<td>TCD_FNC_COPY_DATA_FROM_EPBUF</td>
<td>150</td>
</tr>
<tr>
<td>TCD_FNC_COPY_DATA_TO_EPBUF</td>
<td>150</td>
</tr>
</tbody>
</table>
The HAL sets the `trbLength` element of the TRB header to the total length of the TRB, including the header and any additional parameters that follow the header.

### Function Codes

This section describes the various function codes and the TRBs that are used to convey them to the TCD.

#### 6.5.1 Attaching the TCD

During the attachment process, the HAL calls the single entry point in the TCD with the function code `TCD_FNC_ATTACH`. It calls the single entry point by passing the following TRB:

```c
typedef struct trb_attach {
    TRB_HEADER header;
    pVOID tcdParam;
    USB_HAL_ISR_CALLBACK usbHalIsr;
    pVOID usbHalIsrParam;
    pUSBHAL_DEVICE_INFO pHalDeviceInfo;
} TCD_FNC_ATTACH;
```

<table>
<thead>
<tr>
<th>Function Code</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCD_FNC_INTERRUPT_STATUS_GET</td>
<td>151</td>
</tr>
<tr>
<td>TCD_FNC_INTERRUPT_STATUS_CLEAR</td>
<td>151</td>
</tr>
<tr>
<td>TCD_FNC_ENDPOINT_INTERRUPT_STATUS_GET</td>
<td>152</td>
</tr>
<tr>
<td>TCD_FNC_ENDPOINT_INTERRUPT_STATUS_CLEAR</td>
<td>154</td>
</tr>
<tr>
<td>TCD_FNC_HANDLE_DISCONNECT_INTERRUPT</td>
<td>154</td>
</tr>
<tr>
<td>TCD_FNC_HANDLE_RESET_INTERRUPT</td>
<td>154</td>
</tr>
<tr>
<td>TCD_FNC_HANDLE_RESUME_INTERRUPT</td>
<td>154</td>
</tr>
<tr>
<td>TCD_FNC_HANDLE_SUSPEND_INTERRUPT</td>
<td>154</td>
</tr>
</tbody>
</table>

Table 6.1 TCD Function Requests
The `tcdParam` element points to the TCD-defined parameters structure that the application in the configlette passes to `usbTargTcdAttach()` (see TCD-defined Parameters, p.106). The `usbHalIsr()` is the interrupt service routine defined in the HAL. The interrupt service routine of the TCD should call this routine, passing it the `usbHalIsrParam`, whenever an interrupt event occurs.

This callback uses the following prototype:

```c
typedef (*USB_HAL_ISR_CALLBACK)(pVOID pHALData);
```

The TCD sets the `uNumberEndpoints` element of the `pHalDeviceInfo` structure (shown below) to the number of endpoints supported by the target controller.

```c
typedef usbhal_device_info
{
  UINT8 uNumberEndpoints;
} USBHAL_DEVICE_INFO, *pUSBHAL_DEVICE_INFO;
```

The target application learns details about the device by means of the `USB_APPLN_DEVICE_INFO` structure, shown below:

```c
typedef struct usb_appln_device_info
{
  UINT32 uDeviceFeature; /* bitmap giving feautres supported */
  UINT32 uEndpointNumberBitmap; /* bitmap giving endpoint numbers */
} USB_APPLN_DEVICE_INFO, *pUSB_APPLN_DEVICE_INFO;
```

The TCD populates this structure when the single access point of the TCD is called with the `TCD_FNC_ATTACH` code, and the HAL and target layer pass this structure back to the target application.

In this structure, the `uDeviceFeature` is a bitmap that explains whether the device is USB 2.0 compliant, whether it supports Remote Wakeup, and whether it supports Test Mode. The TCD sets this bitmap to a combination of the bits `USB_FEATURE_DEVICE_REMOTE_WAKEUP`, `USB_FEATURE_TEST_MODE`, and `USB_FEATURE_USB20`. For instance:

```c
uDeviceFeature = (USB_FEATURE_DEVICE_REMOTE | USB_FEATURE_USB20)
```

The TCD sets `uEndpointNumberBitmap` to indicate the endpoints supported by the hardware. The first 16 bits indicate the OUT endpoint and the next 16 bits indicate the IN endpoint, with bits 0 to 15 corresponding to OUT endpoints 0 to 15, and bits 15 to 31 corresponding to IN endpoints 0 to 15.

During the TCD attachment process, the TC should set the various register of the target controller with appropriate values to initialize the target controller.
When the TCD detects certain USB events, it notifies the HAL which in turn calls the management event target application callback that the target layer registered with the HAL (see 5.5.1 mngmtFunc() Callback, p.113). When TCD attachment is complete, the TCD returns the following values:

- a TCD-defined handle in the handle member of the TRB
- the number of endpoints supported by the target controller in the pHalDeviceInfo member of the TRB
- some device specific information required by the target application in the pDeviceInfo member of the TRB

6.5.2 Detaching the TCD

While detaching the TCD, the HAL calls the single entry point of the TCD with the code TCD_FNC_DETACH. The TRB, shown below, is used for this purpose.

```c
typedef struct trb_detach
{
    TRB_HEADER header;        /* TRB header */
} TRB_DETACH, *pTRB_DETACH;
```

The TCD responds to a TCD_FNC_DETACH request by disabling the target controller hardware and releasing all resources that it has allocated on behalf of the controller.

6.5.3 Enabling and Disabling the TCD

In order to enable or disable the target controller the HAL calls the single entry point of the TCD with the code TCD_FNC_ENABLE or TCD_FNC_DISABLE. In response to a TCD_FNC_ENABLE request, the TCD enables the target controller, making the peripheral visible on the USB (if it is connected to a USB).

In response to a TCD_FNC_DISABLE request, the TCD disables the target controller, in effect making the peripheral disappear from the USB to which it is attached. This function is typically requested only if the target application is in the process of shutting down.

Both of these function codes share the same TRB. This TRB is shown below:

```c
typedef struct trb_enable_disable
{
    TRB_HEADER header;        /* TRB header */
} TRB_ENABLE_DISABLE, *pTRB_ENABLE_DISABLE;
```
6.5.4 Setting the Address

The device must set its address to 0 (on bus reset) or to the address specified by the USB host during the SET_ADDRESS request.

To set the address, the HAL calls the single entry point of the TCD with the code TCD_FNC_ADDRESS_SET. This function uses the TRB shown below:

```c
typedef struct trb_address_set {
    TRB_HEADER header;            /* TRB header */
    UINT16 deviceAddress;         /* IN: new device address */
} TRB_ADDRESS_SET, *pTRB_ADDRESS_SET;
```

Once the USB host enumerates the peripheral, the USB host assigns it a new address. The TCD responds to the TCD_FNC_ADDRESS_SET code by setting the target controller to begin responding to the new USB device address as specified by `deviceAddress`.

6.5.5 Resuming the Signal

The USB device may go into a SUSPEND state and stop responding to requests from the host. To make it respond to the host requests the HAL calls the single entry point of the TCD with the function code TCD_FNC_RESUME. The HAL uses the TRB shown in below for this purpose.

```c
typedef struct trb_signal_resume {
    TRB_HEADER header;            /* TRB header */
} TRB_SIGNAL_RESUME, *pTRB_SIGNAL_RESUME;
```

If the USB is not currently in the SUSPEND state, this request has no effect.

6.5.6 Setting and Clearing the Device Feature

The USB host makes the standard requests SET_FEATURE and CLEAR_FEATURE in order to set or clear device-specific features.

In response to these calls, the HAL calls the single entry point of the TCD with the function codes TCD_FNC_DEVICE_FEATURE_SET and TCD_FNC_DEVICE_FEATURE_CLEAR. These share TRB shown below:

```c
typedef struct trb_device_feature_set_clear {
    TRB_HEADER header;
    UINT16 uFeatureSelector;
    UINT8 uTestSelector;
} TRB_DEVFEATURE_SET_CLEAR, *pTRB_DEVFEATURE_SET_CLEAR;
```
The \texttt{uFeatureSelector} element specifies the Remote Wakeup feature or the Test Mode feature (see Table 5-4 for a list of feature selector values). The \texttt{uTestSelector} element specifies the Test Selector values (see Table 5-9 for a list of test selector values).

6.5.7 Getting the Current Frame Number

It is sometimes necessary for clients to re-synchronize with devices when the two are exchanging data isochronously. This function allows a client to query a reference frame number maintained by the device (indicated by \texttt{frameNo}). Please refer to the USB 2.0 specification for more detail.

The TRB that the HAL uses when calling the single entry point of the TCD with the \texttt{TCD_FNC_CURRENT_FRAME_GET} code is shown below:

```c
typedef struct trb_current_frame_get {
    TRB_HEADER header;
    UINT16 frameNo;
} TRB_CURRENT_FRAME_GET, *pTRB_CURRENT_FRAME_GET;
```

In response to the \texttt{TCD_FNC_CURRENT_FRAME_GET} request, the TCD returns the current USB frame number in the \texttt{frameNo} element of the TRB.

6.5.8 Assigning the Endpoints

Before any data is transferred to or from the endpoints, the endpoints must be created. The target layer creates the control endpoints during the bus reset process. The target application creates the generic endpoints during the enumeration process from the host.

The target application always creates the endpoints corresponding to the endpoint numbers that the HAL receives from the target layer in the endpoint descriptor (see Endpoint Descriptor, p.128). The HAL maintains all the endpoint-specific information in the \texttt{USBHAL_PIPE_INFO} structure, shown below:

```c
typedef struct usbhal_pipe_info {
    UINT8 uEndpointAddress;
    UINT8 uTransferType;
    UINT32 pipeHandle;
    LIST_HEAD listHead;
    MUTEX_HANDLE mutexHandle;
} USBHAL_PIPE_INFO, *pUSBHAL_PIPE_INFO;
```
The HAL calls the single entry point of the TCD with the TRB shown below:

```c
typedef struct trb_endpoint_assign {
    TRB_HEADER header;
    pUSB_ENDPOINT_DESCR pEndpointDesc;
    UINT32 uConfigurationValue;
    UINT32 uInterface;
    UINT32 uAltSetting;
    UINT32 pipeHandle;
} TRB_ENDPOINT_ASSIGN, *pTRB_ENDPOINT_ASSIGN;
```

and the function code TCD_FNC_ENDPOINT_ASSIGN (see 5.6.1 Creating and Destroying the Pipes, p.127). The TCD responds to this request by returning to the HAL, using the `pipeHandle` element of the TRB, a handle to the pipe created. The HAL uses this handle for any subsequent request through that pipe.

The target application sets the endpoint descriptor, `pEndpointDesc`; see Endpoint Descriptor, p.128.

### 6.5.9 Releasing the Endpoints

In order to release the endpoint, the HAL calls the single entry point of the TCD with the code TCD_FNC_ENDPOINT_RELEASE and a pipe handle that represents the pipe to release and the endpoint to free. The TRB that the HAL passes to the single entry point of the TCD for this purpose takes the form shown below:

```c
typedef struct trb_endpoint_release {
    TRB_HEADER header;
    UINT32 pipeHandle;
} TRB_ENDPOINT_RELEASE, *pTRB_ENDPOINT_RELEASE;
```

In response to this function call, the TCD releases the particular endpoint and removes the pipe referenced by `pipeHandle`.

### 6.5.10 Setting the Endpoint Status

In order to set the status of the endpoint, the HAL calls the single entry point of the TCD with the code TCD_FNC_ENDPOINT_STATE_SET in the TRB shown below:

```c
typedef struct trb_endpoint_state_set
```
The state element is either TCD_ENDPOINTSTALL or TCD_ENDPOINTUNSTALL.

6.5.11 Getting the Endpoint Status

The host may issue a GET_STATUS request to get the status of an endpoint.

In order to get the status of the endpoint, the HAL calls the single entry point of the TCD with the function code TCD_FNC_ENDPOINT_STATUS_GET in the TRB shown below:

```c
typedef struct trb_endpoint_status_get {
    TRB_HEADER header;
    UINT32 pipeHandle;
    pUINT16 pStatus;
} TRB_ENDPOINT_STATUS_GET, *pTRB_ENDPOINT_STATUS_GET;
```

The TCD returns the status of the endpoint (see Table 5-7 for a list of valid status values and their meanings).

6.5.12 Submitting and Canceling ERPs

Once the pipes are created, they may be used to transfer data (see 5.6.2 Transferring and Aborting Data, p.130). The HAL calls the single entry point of the TCD with the code TCD_FNC_COPY_DATA_FROM_EPBUF or TCD_FNC_COPY_DATA_TO_EPBUF depending on the direction of the pipe for which the request is issued by the target layer.

These function codes use the TRBs shown below:

```c
typedef struct trb_copy_data_from_epbuf {
    TRB_HEADER header;
    UINT32 pipeHandle;
    p UCHAR pBuffer;
    UINT32 uActLength;
} TRB_COPY_DATA_FROM_EPBUF, *pTRB_COPY_DATA_FROM_EPBUF;
```

```c
typedef struct trb_copy_data_to_epbuf {
    TRB_HEADER header;
} TRB_COPY_DATA_TO_EPBUF, *pTRB_COPY_DATA_TO_EPBUF;
```
6 Target Controller Drivers
6.5 Function Codes

UINT32 pipeHandle;
*p UCHAR pBuffer;
UINT32 uActLength;
} TRB_COPY_DATA_TO_EPBUF, *pTRB_COPY_DATA_TO_EPBUF;

The pipeHandle element refers to the endpoint to whose or from whose FIFO buffer the data is to be copied. The data is copied to or from the pBuffer.

The uActLength element is set by the HAL before it submits the TRB, indicating the size of the buffer to be transmitted. However, the actual amount of data transmitted through the TCD may be different, depending on the maximum packet size for the endpoint. If the TCD transmits a different amount of data than is indicated by uActLength, the TCD should overwrite uActLength with the actual amount of data that was transferred before returning from its single entry point.

6.5.13 Determining Whether the Buffer is Empty

The HAL calls the single entry point of the TCD with the function code TCD_FNC_IS_BUFFER_EMPTY to see whether the FIFO buffer associated with the pipe designated by pipeHandle is empty or not. This function code uses the TRB shown below:

typedef struct trb_is_buffer_empty
{
   TRB_HEADER header;
   UINT32 pipeHandle;
   BOOL bufferEmpty;
} TRB_IS_BUFFER_EMPTY, *pTRB_IS_BUFFER_EMPTY;

6.5.14 Getting and Clearing Interrupts

The HAL calls the single entry point of the TCD with the codes TCD_FNC_INTERRUPT_STATUS_GET or TCD_FNC_INTERRUPT_STATUS_CLEAR in order to get or clear the interrupt status. These codes share the TRB shown below:

typedef struct trb_interrupt_status_get_clear
{
   TRB_HEADER header;
   UINT32 uInterruptStatus;
} TRB_INTERRUPT_STATUS_GET_CLEAR, *pTRB_INTERRUPT_STATUS_GET_CLEAR;

In response to the TCD_FNC_INTERRUPT_STATUS_GET, the TCD indicates the types of interrupts that are pending by setting the uInterruptStatus element of the TRB. This element is a bitmap with the following form:
- bit 0 – disconnect interrupt
- bit 1 – reset interrupt
- bit 2 – suspend interrupt
- bit 3 – resume interrupt
- bit 4 – endpoint interrupt

In response to the TCD_FNC_INTERRUPT_STATUS_CLEAR, the TCD clears the indicated interrupts. It uses the same bitmap format to indicate which interrupts to clear.

Figure 6-1 illustrates how interrupts are handled:

6.5.15 Retrieving an Endpoint-Specific Interrupt

If the TCD indicates to the HAL that an endpoint interrupt has occurred, the HAL calls the single entry point of the TCD with the function code TCD_FNC_ENDPOINT_INTERRUPT_STATUS_GET to determine the specific type
of interrupt that has occurred on the endpoint. The TRB for this process is shown below:

```c
typedef struct trb_endpoint_interrupt_status_get {
    TRB_HEADER header;
    UINT32 pipeHandle;
    UINT32 uEndptInterruptStatus;
} TRB_ENDPOINT_INTERRUPT_STATUS_GET, *pTRB_ENDPOINT_INTERRUPT_STATUS_GET;
```

**uEndpointInterruptStatus** is a bitmap that indicates the types of interrupt that have occurred on that endpoint. It takes the following form:

- bit 0 – endpoint Interrupt
- bit 1 – setup Interrupt
- bit 2 – OUT interrupt
- bit 3 – IN interrupt

The TCD uses bits 4 to 11 to report errors. These errors are defined in `usbTcd.h` and are shown in **Table 6-2**:

<table>
<thead>
<tr>
<th>Error</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBTCD_ENDPOINT_TRANSFER_SUCCESS</td>
<td>0x0000</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_DATA_TOGGLE_ERROR</td>
<td>0x00E0</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_PID_MISMATCH</td>
<td>0x00F0</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_COMMUN_FAULT</td>
<td>0x0100</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_STALL_ERROR</td>
<td>0x0110</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_DATA_OVERRUN</td>
<td>0x0120</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_DATA_UNDERRUN</td>
<td>0x0130</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_CRC_ERROR</td>
<td>0x0140</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_TIMEOUT_ERROR</td>
<td>0x0150</td>
</tr>
<tr>
<td>USBTCD_ENDPOINT_BIT_STUFF_ERROR</td>
<td>0x0160</td>
</tr>
</tbody>
</table>
6.5.16 Clearing All Endpoint Interrupts

The HAL calls the single entry point of the TCD with the code `TCD_FNC_ENDPOINT_INTERRUPT_STATUS_CLEAR` to clear all of the endpoint-specific interrupts. The TRB for this process is shown below:

```c
typedef struct trb_endpoint_interrupt_status_clear {
    TRB_HEADER header;
    UINT32 pipeHandle;
} TRB_ENDPOINT_INTERRUPT_STATUS_CLEAR,
    *pTRB_ENDPOINT_INTERRUPT_STATUS_CLEAR;
```

6.5.17 Handling Disconnect, Reset, Resume, and Suspend Interrupts

When disconnect, suspend, resume, and reset events happen, some hardware may have to handle the event by doing something specific to its register set. The HAL calls the single entry point of the TCD with the following codes in order to handle these events:

- `TCD_FNC_HANDLE_DISCONNECT_INTERRUPT`
- `TCD_FNC_HANDLE_RESET_INTERRUPT`
- `TCD_FNC_HANDLE_RESUME_INTERRUPT`
- `TCD_FNC_HANDLE_SUSPEND_INTERRUPT`

The TCD only uses these notices if it must do something specific in response to these interrupts (otherwise the TCD just returns from its single entry point without doing anything). On a disconnect, for example, some target controllers expect some registers to be reset—this can be done in the disconnect interrupt handler.

The TRB that the HAL constructs for the `TCD_FNC_HANDLE_DISCONNECT_INTERRUPT` event is shown below:

```c
typedef struct trb_handle_disconnect_interrupt {
    TRB_HEADER header;
} TRB_HANDLE_DISCONNECT_INTERRUPT,
    *pTRB_HANDLE_DISCONNECT_INTERRUPT;
```

The TRB that the HAL constructs for the `TCD_FNC_HANDLE_RESET_INTERRUPT` event is shown below:

```c
typedef struct trb_handle_reset_interrupt {
    TRB_HEADER header;
} TRB_HANDLE_RESET_INTERRUPT,
    *pTRB_HANDLE_RESET_INTERRUPT;
```
The TRB that the HAL constructs for the `TCD_FNC_HANDLE_RESUME_INTERRUPT` event is shown below:

typedef struct trb_handle_resume_interrupt
{
    TRB_HEADER header;
} TRB_HANDLE_RESUME_INTERRUPT, *pTRB_HANDLE_RESUME_INTERRUPT;

The TRB that the HAL constructs for the `TCD_FNC_HANDLE_SUSPEND_INTERRUPT` event is shown below:

typedef struct trb_handle_suspend_interrupt
{
    TRB_HEADER header;
} TRB_HANDLE_SUSPEND_INTERRUPT, *pTRB_HANDLE_SUSPEND_INTERRUPT;
7.1 Introduction

The Wind River USB host and peripheral stacks are designed to ease porting across various Wind River BSP platforms.

The USB host stack is successfully ported and tested on both:
- PCI Buses
- Non-PCI Buses

The USB peripheral stack is successfully ported and tested over the PCI bus only. It can also be easily ported on a non-PCI Bus with minimal effort.

The Wind River USB host and peripheral stacks have been successfully tested on both big- and little-endian platforms and across several Wind River BSPs. See your product release notes for a list of tested devices.
For all the host controllers, the **usbPciLib** module (that is, the **usbPciStub.c** file for the corresponding BSP) should be modified to use the correct PCI or non-PCI routines in the underlying BSP. See your product release notes for the list of supported BSPs.

### 7.2 Configuring USB Host Stack Hardware

As shipped, the Wind River USB host stack supports EHCI, OHCI, and UHCI USB host controllers. The USB hardware is initialized and enabled during the USB initialization. The following sections discuss in detail how to initialize the USB hardware and port the USB host stack on both PCI and non-PCI buses.

#### 7.2.1 Initialization of USB Hardware

The USB initialization routine, **usrUsbInit()**, defined in `installDir/vxworks-6.x/target/config/comps/src/usrUSBInit.c`, calls the USB hardware configlette routine, **usb2PciInit()**, defined in `installDir/vxworks-6.x/target/config/comps/src/usb2PciInit.c`. This sequence initializes the resources for various PCI- and non-PCI-based host controllers for the different types of EHCI, OHCI, and UHCI host controllers.

The configlette routines for various host controllers first initialize the non-PCI-based host controllers, if any, and then the PCI-based host controllers.

#### Example 7-1 Initializing Resources for OHCI Controllers

The following sample code demonstrates how to initialize the resources for the PCI- and non-PCI-based OHCI controllers:

```c
void usb2OhciPciInit()
{
    UINT16 data16;
    UINT8 PCIBusNumber;
    UINT8 PCIDeviceNumber;
    UINT8 PCIFunctionNumber;
    PCI_CFG_HEADER pciCfgHdr;
    USB_HCD_BUS_INFO hcdBusInfo;
    UINT12 nonPciOhciCount = 0;

    while (maxOhciCount < MAX_NO_OF_OHCI_CONTROLLERS)
    {
```

158
/* locate the Non-Pci host controllers
 * Call the usbPciClassFind routine with
 * USB_IF_NON_PCI as programming interface
 */

if (!usbPciClassFind ((OHCI_CLASS), OHCI_SUBCLASS,
    OHCI_PGMIF | USB_IF_NON_PCI, maxOhciCount, (pUINT8)
    &hcdBusInfo, NULL, NULL))
    {
    /* No non-PCI OHCI Controller found */
    break;
    }

/* usbPciClassFind routine populates the hcdBusInfo
 * structure with appropriate values.
 * This structure is used to allocate the resource
 * for hardware initialization of OHCI Controller
 */

......

/* Now initialize the PCI-based Controller */

if (!usbPciClassFind (OHCI_CLASS, OHCI_SUBCLASS, OHCI_PGMIF,
    maxOhciCount, &PCIBusNumber, &PCIDeviceNumber,
    &PCIFunctionNumber))
    {
    /* No PCI OHCI Controller found */
    break;
    }

/* If any PCI-based host controller is found,
 * obtain the PCI Configuration header to obtain
 * the base address and the interrupt line.
 * Allocate the function pointers for the host controllers.
 */

Similarly, the configuration routines for EHCI and UHCI initialize the EHCI and
UHCI host controllers, respectively.

### 7.2.2 USB_HCD_BUS_INFO Data Structure

The **USB_HCD_BUS_INFO** data structure defines the function pointers used for
performing host bus specific operations. It is populated for both PCI- and
non-PCI-based controllers, and is defined in
installDir/vxworks-6.x/target/h/usb/pciConstants.h, as follows:

typedef struct usbHcdBusInfo
{
    FUNCPTR pFuncDataSwap;
    FUNCPTR pFuncCpuToBus;
}
pFuncBusToCpu
Used for converting the bus specific address to CPU address.

The programmer should register this function pointer with the function that performs the conversion from bus specific address to CPU address.

pFuncCpuToBus
Used for converting the CPU address to the bus specific address.

The programmer should register this function pointer with the function that performs the conversion from CPU address to bus specific address.

pFuncIntConnect
Used for hooking the ISR to an interrupt line.

- For PCI-Based Controllers: the programmer should register this function pointer with `usbPciIntConnect()` implemented in the `usbPciStub.c`.
- For Non-PCI-Based Controllers: the programmer should register this function pointer with appropriate function depending upon the underlying BSP.

pFuncDataSwap
Used for register read/write conversions and the data structure conversions.

The programmer should register this function pointer with the function `usbDataSwap()` as follows:

- For PCI-Based Controllers: this function should swap the contents for the big-endian architectures. For little-endian architectures, this function pointer should be set to NULL.
- For Non-PCI-Based Controllers: this function should swap the contents depending on the underlying BSP.

pFuncBufferSwap
Used for the data structure conversions.

The programmer should register this function pointer with the function `usbDataSwap()` as follows:

- For PCI-Based Controllers: this function should swap the contents for the big-endian architectures. For little-endian architectures, this function pointer should be set to NULL.
- For Non-PCI-Based Controllers: this function should swap the contents depending on the underlying BSP.
**pFuncIntDisconnect**
Used for disconnecting an ISR to an interrupt line.

- For PCI-Based Controllers: the programmer should register this function pointer with `usbPciIntRestore()` implemented in the `usbPciStub.c`.
- For Non-PCI-Based Controllers: the programmer should register this function pointer with appropriate function depending upon the underlying BSP.

**pFuncBufferSwap**
Used for swapping the contents of the buffers.

The programmer should register this function pointer with the function `usbBufferSwap()` as follows:

- For PCI-Based Controllers: this function pointer should be always set to NULL, regardless of the endianess of the architecture.
- For Non-PCI-Based Controllers: this function should swap the buffer depending on the underlying BSP.

For some non-PCI host controllers that follow big-endian format, `usbBufferSwap()` can be registered as the function pointer for `pFuncBufferSwap`. However, it depends solely on hardware and may not be necessary in all big-endian BSPs. For example, the on-board OHCI host controller on ms7727se needs to have buffers swapped. However, the on-board OHCI host controller on pb1000 does not need to have buffers swapped. For sample code for this function, see *Data Conversion Routines*, p.171.

**pBaseAddress**
Used to hold the base address for the controller.

- For PCI-Based Controllers: this value is obtained by reading the PCI configuration space.
- For Non-PCI Controllers: this is a hard-coded value which can be obtained from the underlying BSP.

**Irq**
Holds the interrupt vector number for the USB host controller:

- For PCI-Based Controllers: this value is obtained by reading the PCI configuration space.
- For Non-PCI Controllers: this is a hard-coded value that can be allocated from the underlying BSP.
pDeviceSpecificData
Can be used for storing device specific data.

### 7.2.3 Locating the Host Controller

The `usbPciClassFind()` routine is used to locate both the PCI- and non-PCI-based host controllers. This routine is defined in `installDir/vxworks-6.x/target/config/BSP/usbPciStub.c` as follows:

```c
********************************************************************
* usbPciClassFind - locates a PCI or Non-PCI host controller       *
*                                                                   *
********************************************************************
BOOL usbPciClassFind
{
    UINT8 pciClass,
    UINT8 subClass,
    UINT8 pgmIf,        /* programming interface */
    UINT16 index,
    pUINT8 pBusNo,
    pUINT8 pDeviceNo,
    pUINT8 pFuncNo
}

The third parameter of the function specifies the programming interface.

In the case of PCI-based OHCI host controller, the value of 0x10 (OHCI_PGMIF) is passed as the third parameter. This routine should call the `pciFindClass()` routine to obtain the PCI Bus Number, PCI Device Number, and the PCI Function Number for the PCI host controller. These values are further used to obtain the PCI Header Configuration. Similarly for EHCI and UHCI host controllers, the value for the programming interface should be 0x20 (EHCI_PGMIF) and 0x00 (UHCI_PGMIF), respectively.

**NOTE:** Do not confuse the two distinct routines that have similar names and are discussed in this chapter: `usbPciClassFind()` and `pciFindClass()`.

However, in the case of non-PCI host controllers, the programming interface values for different host controllers (EHCI_PGMIF, OHCI_PGMIF, and UHCI_PGMIF) should be ORed with 0x80 (USB_IF_NON_PCI).

If any non-PCI controllers are found they should populate the function pointers members of the `usbHcdBusInfo` structure.
The macros for the various programming interfaces are defined in the header file
installDir/vxworks-6.x/target/h/usb/pciConstants.h.

The **usbPciClassFind()** routine should do the following, depending on the type of
host controller.

- If the programming interface is for non-PCI-based host controllers, then:
  - Check if the **usbHcdBusInfo** structure is allocated. If it is not, return FALSE.
  - Populate the function pointers members of the **usbHcdBusInfo**
    structure as explained in the sections in 7.2.1 Initialization of USB
    Hardware, p.158 and 7.2.2 USB_HCD_BUS_INFO Data Structure, p.159
    for non-PCI host controllers.
  - Set the members **pBaseAddress** and **Irq** of the **usbHcdBusInfo**
    structure with appropriate base address and irq number.
  - Return TRUE.

- If the programming interface is for PCI-based host controllers:
  - Call the function **pciFindClass()** to obtain the PCI Bus No., PCI
    Device No. and the PCI Function No.
  - Return TRUE.

- Otherwise:
  - Return FALSE.

**Example 7-2  Pseudocode for usbPciClassFind()**

```c
BOOL usbPciClassFind (/* parameters */ )
{
  . . .
  . . .
  if ((pgmIf & USB_IF_NON_PCI) == USB_IF_NON_PCI)
  {
    /* non-pci controller found. */
    /* Populate the usbHcdBusInfo structure */
    . . . . . . .
    return TRUE;
  }

  else if ((pgmIf & XHCI_PGMIF) == XHCI_PGMIF)
  {
    /* call the pciFindClass routine to find */
    /* the PCI-based host controller */
```

```c
```
7.2.4 Porting for PCI Buses

As with any PCI device, the EHCI, OHCI, and UHCI host controllers need to be configured to use specific PCI resources.

A PCI UHCI host controller, for example, uses a single I/O address range and one interrupt channel. In contrast, a PCI OHCI controller and a PCI EHCI controller use a single memory address range and one interrupt channel. All of the types of controllers need to be assigned a unique address range and an interrupt channel. The base address and the interrupt line for the PCI-based controllers are obtained by reading the PCI configuration space.

In order to port any PCI-based USB host controller on any of the architectures, you need to do the following:

1. Determine the correct PCI base address.
2. Determine the correct IRQ line.
3. Map the virtual address generated by the CPU with the PCI physical address.

Example 7-3 Initializing OHCI Controllers

Following is a sample code for initialization of PCI-based OHCI host controller:

```c
/**************************************************************************
* usb2OhciPciInit - for OHCI host bus specific initialization
* RETURNS: none
*/

void usb2OhciPciInit()
{
    ..
    ..
```

NOTE: The configlette routines for various host controllers are written to search first for the non-PCI-based host controllers, and then for the PCI-based controllers. This approach thereby handles BSPs that contain both types of host controllers.
/* Initialize the PCI-based host controller */

while ( maxOhciCount < MAX_NO_OF_OHCI/controllers )
{
    /* Find the OHCI Controller */
    if ( !usbPciClassFind( OHCI_CLASS, OHCI_SUBCLASS, OHCI_PGMIF, 
        hostControlerInex, &PCIBusNumber, &PCIDeviceNumber, 
        &PCIFunctionNumber))
    {
        break;
    }

    /* Obtain the PCI configuration header */
    usbPciConfigHeaderGet( PCIBusNumber, PCIDeviceNumber, 
        PCIFunctionNumber, &pciCfgHdr);

    /* Populate the function pointers */
    pOhciBusInfo[maxOhciCount]->pFuncDataSwap = (FUNCPTR)usbDataSwap;
    pOhciBusInfo[maxOhciCount]->pFuncBufferSwap = NULL;
    pOhciBusInfo[maxOhciCount]->pFuncCpuToBus = 
        (FUNCPTR)usbMemToPci;
    pOhciBusInfo[maxOhciCount]->pFuncBusToCpu = 
        (FUNCPTR)usbPciToMem;
    pOhciBusInfo[maxOhciCount]->pFuncIntConnect = 
        (FUNCPTR)usbPciIntConnect;
    pOhciBusInfo[maxOhciCount]->pFuncIntDisconnect = 
        (FUNCPTR)usbPciIntRestore;

    /* Update the base address */
    pOhciBusInfo[maxOhciCount]->pBaseAddress = 
        (pVOID)pciCfgHdr.baseReg[0];

    /* Update the IRQ */
    pOhciBusInfo[maxOhciCount]->irq = 
        pciCfgHdr.intLine - INT_NUM_IRQ0;
    maxOhciCount++;
}

NOTE: Similarly we can locate and populate function pointers for EHCI- and 
UHCI-based host controllers.
7.2.5 Endian Conversion for PCI Buses

The data sent through 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 host should be converted from big-endian to little-endian in the BSP. Conversely, any data passed from the host to the target must be converted from little-endian to big-endian format. The endianess conversion is handled in the configlette routine. No action is required in the BSP for endian conversion for PCI-based controllers.

7.2.6 Porting for Non-PCI Buses

One should keep a note of the following points while porting the Wind River USB host stack 2.2 over the non-PCI buses:

- The HCD does not know the host bus to which the host controller is connected; thus, all changes with respect to non-PCI porting should be isolated in target/config/BSP/usbPciStub.c.
- The USB 2.0 specification conforms to little-endian format, whereas the reads and writes over the non-PCI bus may be in either little-endian or in big-endian format, depending on the underlying architecture. The required changes must be done in the BSP in order to handle the endian conversion.

Assignment of Non-PCI Resources

The assignment of non-PCI resources is done by the configlette routine. The configlette routine is defined in the file

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

The configlette routine first searches for non-PCI-based host controllers and then for PCI-based controllers.

Example 7-4 Initializing Resources for OHCI Controllers

The following sample code is used to initialize the resources for a non-PCI OHCI controller:

```c
void usb2OhciPciInit()
{
    ...
    ...

    USB_HCD_BUS_INFO hdcBusInfo;
```
/* number of non-PCI OHCI controllers */
UINT32 nonPciOhciCount = 0;

/*
 * This loop initializes the array for all the
 * non-PCI OHCI host controllers in the host system.
 */
while (maxOhciCount < MAX_NO_OF_OHCI_CONTROLLERS)
{
    /* Find the non-PCI OHCI Controller */
    if (!usbPciClassFind ((OHCI_CLASS), OHCI_SUBCLASS,
                         OHCI_PGMIF | USB_IF_NON_PCI, maxOhciCount,
                         (pUINT8) &hcdBusInfo, NULL, NULL))
        {
            /* No non-PCI OHCI Controller found */
            break;
        }
    /* Allocate memory for the bus Info data structure */

    /* Copy all the members to the memory allocated structure */
    pOhciBusInfo[maxOhciCount]->pFuncDataSwap =
        hcdBusInfo.pFuncDataSwap;
    pOhciBusInfo[maxOhciCount]->pFuncBufferSwap =
        hcdBusInfo.pFuncBufferSwap;
    pOhciBusInfo[maxOhciCount]->pFuncCpuToBus =
        hcdBusInfo.pFuncCpuToBus;
    pOhciBusInfo[maxOhciCount]->pFuncBusToCpu =
        hcdBusInfo.pFuncBusToCpu;
    pOhciBusInfo[maxOhciCount]->pBaseAddress =
        hcdBusInfo.pBaseAddress;
    pOhciBusInfo[maxOhciCount]->irq = hcdBusInfo.irq;
    pOhciBusInfo[maxOhciCount]->pFuncIntConnect =
        hcdBusInfo.pFuncIntConnect;
    pOhciBusInfo[maxOhciCount]->pFuncIntDisconnect =
        hcdBusInfo.pFuncIntDisconnect;
    maxOhciCount++;
}

/* Store the maximum number of non-PCI host controllers */
nonPciOhciCount = maxOhciCount;
/* Now initialize the PCI-based Controller */
...
Therefore, in the `usbPciClassFind()` routine, the programmer should do the following for non-PCI support:

1. Check if the `usbHcdBusInfo` structure is allocated. If not, return FALSE.
2. Check if the programming interface is `USB_IF_NON_PCI` (0x80). If not, return FALSE and look for PCI-based resources depending upon the BSP.
3. If the programming interface is `USB_IF_NON_PCI` (0x80), populate the pointer to `usbHcdBusInfo` structure, as explained in 7.2.1 Initialization of USB Hardware, p.158 and 7.2.2 USB_HCD_BUS_INFO Data Structure, p.159, and return TRUE.

Example 7-5 Example usbPciClassFind() Routine for OHCI Controller

Following is the sample `usbPciClassFind()` routine written for ms7727se big-endian BSP having a non-PCI-based OHCI host controller:

```c
/**************************************************************************
* usbPciClassFind - Locates PCI or Non PCI controllers by class.
* *
**************************************************************************

BOOL usbPciClassFind
(
    UINT8 pciClass, /* PCI device class */
    UINT8 subClass, /* PCI device sub-class */
    UINT8 pgmIf, /* Programming interface */
    UINT16 index, /* Caller wants nth matching dev */
    pUINT8 pBusNo, /* Bus number of matching dev */
    pUINT8 pDeviceNo, /* Device number of matching dev */
    pUINT8 pFuncNo /* Function number of matching dev */
)
{
    pUSB_HCD_BUS_INFO pHcdBusInfo = (pUSB_HCD_BUS_INFO)pBusNo;

    if (pHcdBusInfo == NULL)
        return FALSE;

    if (((pgmIf & USB_IF_NON_PCI) != USB_IF_NON_PCI) || (index != 0))
        return FALSE;

    pHcdBusInfo->pFuncDataSwap = NULL;
    pHcdBusInfo->pFuncBufferSwap = NULL;
    pHcdBusInfo->pFuncCpuToBus = usbNonPcCpuToBus;
    pHcdBusInfo->pFuncBusToCpu = usbNonPciBusToMem;
    pHcdBusInfo->baseAddress = (UINT32) USB_HOST_REVR;
    pHcdBusInfo->irq = IV_USB_H;
    pHcdBusInfo->pFuncHwInit = usbNonPciIntConnect;
    pHcdBusInfo->pFuncIntConnect = usbNonPciIntConnect;
```

7.3 Configuring USB Peripheral Stack Hardware

The USB peripheral stack initialization routine, `usrUsbTargXXXInit()`, defined in `installDir/vxworks-6.x/target/config/comps/src`, calls the USB hardware configlette routine, of the corresponding Target Controller, defined in `installDir/vxworks-6.x/target/config/comps/src/usrUsbTargPciInit.c`.

The different hardware initialization routines are:

- `sysIsp1582PciInit`
  Configures the PCI-based Philips ISP1582 Target Controller

- `sys2NET2280PciInit`
  Configures the PCI-based Netchip NET2280 Target Controller

Configuring the USB Peripheral Hardware involves:

- Determining the base address of the controller
- Determining the interrupt request number of the controller
NOTE: Certain target controller like Philips PDIUSBD12 TC have a hard-coded base address and IRQ number. For these controllers, you do not need to do any hardware specific initialization. The base address and IRQ number can be used directly to do various register read and writes.

Example 7-6 Initializing resources for NET2280 controller

The following sample code demonstrates how to initialize the resources for the PCI-based NET2280 target controller:

```c
/******************************************************************************
 * sys2NET2280PciInit - to configure the PCI interface for NET2280
 *
 * This function is used to configure the PCI interface for NET2280. It
 * obtains the PCI Configuration Header for the NET2280 and provides
 * with the base addresses and the irq number.
 */

void sys2NET2280PciInit (void)
{
    /* locate the NET2280 controller by passing the vendor id and
     * device id
     */
    USB_PCI_FIND_DEVICE(NET2280_VENDOR_ID, NET2280_DEVICE_ID,
                        &nDeviceIndex, &PCIBusNumber, &PCIDeviceNumber,
                        &PCIFunctionNumber);

    /* If any NET2280 target controller is found,
     * read the PCI Configuration header to obtain
     * the base address and the interrupt line.
     */
    /* Get the configuration header */
    usbPciConfigHeaderGet (PCIBusNumber, PCIDeviceNumber,
                           &PCIFunctionNumber, &pciCfgHdr);

    base_address   = pciCfgHdr.baseReg[0];
    interrupt_line = pciCfgHdr.intline;
}
```
7.4 Creating BSP-Specific usbPciLib Stub Files

The BSP-specific `usbPciLib` stub files provide the various interfaces that must be implemented in the `installDir/vxworks-6.x/target/config/BSP/usbPciStub.c` file to give PCI and non-PCI support to USB host/target controllers.

8-bit, 16-bit, and 32-Bit Data I/O

The functions listed below are used by the I/O mapped PCI-based USB host/target controller to read or write 8-, 16-, or 32-bit data over the PCI bus.

- `usbPciByteIn()`
- `usbPciWordIn()`
- `usbPciDwordIn()`
- `usbPciByteOut()`
- `usbPciWordOut()`
- `usbPciDwordOut()`

Address Range Translation Routines

The functions listed below are used to convert the CPU-generated memory address to the physical address and vice versa. The conversion is required for the CPU and the host controller to map each other’s memory addresses.

The programmer should register this function with appropriate function pointer as explained in 7.2.2 USB_HCD_BUS_INFO Data Structure, p.159.

- `usbMemToPci()`
- `usbPciToMem()`
- `usbNonPciCpuToBus()`
- `usbNonPciBusToMem()`

**NOTE:** The non-PCI function definition is only necessary if the BSP supports both PCI and non-PCI USB host/target controllers. If the BSP supports only non-PCI host controllers, the existing interface for PCI that performs the conversion from the PCI address to the CPU address, `usbPciToMem()`, and from the CPU to the PCI address, `usbMemToPci()`, can be used.

Data Conversion Routines

The `usbBufferSwap()` function is used to swap the contents of the user buffer. It should be registered with the `pFuncBufferSwap()` function pointer member of the structure `usbHcdBusInfo`, as described in 7.2.1 Initialization of USB Hardware, p.158.
Below is a sample implementation of the function:

```c
LOCAL VOID usbBufferSwap (  
    unsigned char *pBuffer, /* Pointer to the buffer */  
    UINT32 size /* Size of the contents in buffer */  
)  
{  
    /* buffers always aligned */  
    UINT32 * bufLong = (UINT32 *) pBuffer;  
    int i;  
    for (i = ((size + 3) / 4); i > 0; i--)  
    {  
        *bufLong = TO_LITTLEL(*bufLong);  
        bufLong++;  
    }  
}
```

**Interrupt Routines**

The following routines are used to hook and unhook the interrupt lines with the ISR:

- `usbPciIntConnect()`
- `usbPciIntRestore()`
- `usbNonPciIntConnect()`
- `usbNonPciIntRestore()`

The programmer should register this function with appropriate function pointer as explained in 7.2.1 Initialization of USB Hardware, p.158.

**NOTE:** The non-PCI function definition is only necessary if the BSP supports both PCI- and non-PCI-based USB host/target controllers. If the BSP supports only non-PCI host controllers, the existing interface for the PCI can be used.

Internally, the `usbPciStub.c` files implement the above routines by defining a series of macros. When a new VxWorks project is created, based on a BSP that includes USB host stack support, a template `usbPciStub.c` file is copied from the `target/config/template` directory to the BSP directory. These USB PCI macros can be customized before first using.
A.1 Introduction

This section provides high-level information about the software elements of Wind River USB as it pertains to VxWorks. It includes information such as dependencies, component initialization routines, libraries, and, where applicable, VxWorks component locations and component include macros.

Reference Documentation

To access reference documentation in VxWorks 6.x, see Help > Help Contents > Wind River Documentation > References. In VxWorks 5.x, see docs/libIndex.html.
A.2 Wind River USB Host Stack Components

This section lists the components used for the Wind River USB host stack.

A.2.1 USB Host Stack

Locating This Component

Component Name
USB Host Stack

VxWorks Component Name
USB Host Stack

VxWorks Component Tree Location
hardware > buses > USB Hosts > USB Host Stack

Include Macro Name
#define INCLUDE_USB

Configuration Summary

Init Routine
usbdPciInit()

Configlettes
usrUsbPciInit.c

Dependencies

The USB host stack requires at least one host controller interface component.
A.2.2 Wind River USB Host Stack Initialization

Locating This Component

Component Name
USB Host Stack Initialization

VxWorks Component Name
USB Host Stack Init

VxWorks Component Tree Location
```
hardware > buses > USB Hosts > USB Host Init > USB Host Stack Init
```

Include Macro Name
```
#define INCLUDE_USB_INIT
```

Configuration Summary

Init Routine
```
usbInit()
```

Configlettes
```
usrUsbInit.c
```

Dependencies

Requires INCLUDE_USB. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.3 Enhanced Host Controller Interface

Locating This Component

Component Name
Enhanced Host Controller Interface

VxWorks Component Name
EHCI
VxWorks Component Tree Location
   hardware > buses > USB Hosts > EHCI

Include Macro Name
   #define INCLUDE_EHCI

Dependencies

Requires INCLUDE_USB.

A.2.4 Enhanced Host Controller Interface Initialization

Locating This Component

   Component Name
   EHCI USB Host Controller Driver Initialization

   VxWorks Component Name
   EHCI Init

   VxWorks Component Tree Location
   hardware > buses > USB Hosts > USB Host Init > EHCI Init

   Include Macro Name
   #define INCLUDE_EHCI_INIT

Configuration Summary

   Init Routine
   usrUsbHcdEhciAttach()

   Configlettes
   usrUsbHcdEhciInit.c

Dependencies

Requires INCLUDE_EHCI and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.
A.2.5 Open Host Controller Interface

Locating This Component

- Component Name: Open Host Controller Interface
- VxWorks Component Name: OHCI
- VxWorks Component Tree Location: `hardware > buses > USB Hosts > OHCI`
- Include Macro Name: `#define INCLUDE_OHCI`

Dependencies

- Requires `INCLUDE_USB`.

A.2.6 Open Host Controller Interface Initialization

Locating This Component

- Component Name: Open Host Controller Interface Initialization
- VxWorks Component Name: OHCI Init
- VxWorks Component Tree Location: `hardware > buses > USB Hosts > USB Host Init > OHCI Init`
- Include Macro Name: `#define INCLUDE_OHCI_INIT`
Configuration Summary

Init Routine
usrUsbHcdOhciAttach()

Configlettes
usrUsbHcdOhciInit.c

Dependencies
Requires INCLUDE_OHCI and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.7 Universal Host Controller Interface

Locating This Component

Component Name
Universal Host Controller Interface

VxWorks Component Name
UHCI

VxWorks Component Tree Location
hardware > buses > USB Hosts > UHCI

Include Macro Name
#define INCLUDE_UHCI

Dependencies
Requires INCLUDE_USB.
A.2.8 Universal Host Controller Interface Initialization

Locating This Component

Component Name
Universal Host Controller Interface Initialization

VxWorks Component Name
UHCI Init

VxWorks Component Tree Location
hardware > buses > USB Hosts > USB Host Init > UHCI Init

Include Macro Name
#define INCLUDE_UHCI_INIT

Configuration Summary

Init Routine

usrUsbHcdUhciAttach()

Configlettes

usrUsbHcdUhciInit.c

Dependencies

Requires INCLUDE_UHCI and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.9 USB Keyboard Driver

Locating This Component

Component Name
USB Keyboard Driver

VxWorks Component Name
Keyboard
VxWorks Component Tree Location
    hardware > peripherals > USB Devices > Keyboard

Include Macro Name
    #define INCLUDE_USB_KEYBOARD

Libraries

    usbKeyboardLib

Dependencies

    Requires INCLUDE_USB.

A.2.10 USB Keyboard Driver Initialization

Locating This Component

    Component Name
    USB Keyboard Class Driver Initialization

    VxWorks Component Name
    Keyboard Init

    VxWorks Component Tree Location
    hardware > peripherals > USB Devices > USB Device Init > Keyboard Init

Include Macro Name
    #define INCLUDE_USB_KEYBOARD_INIT

Configuration Summary

    Init Routine
    usrUsbKbdInit()

    Configlettes
    usrUsbKbdInit.c
A Runtime Configuration Summary
A.2 Wind River USB Host Stack Components

Parameters

USB_KBD_QUEUE_SIZE
Maximum bytes of data that can be stored in the keyboard circular buffer. The
default value is 8.

Dependencies

Requires INCLUDE_USB_KEYBOARD and INCLUDE_USB_INIT. Should not be
used with the USB Tool, INCLUDE_USBTOOL.

A.2.11 USB Mouse Driver

Locating This Component

Component Name
USB Mouse Driver

VxWorks Component Name
Mouse

VxWorks Component Tree Location
hardware > peripherals > USB Devices > Mouse

Include Macro Name
#define INCLUDE_USB_MOUSE

Libraries

usbMouseLib

Dependencies

Requires INCLUDE_USB.
A.2.12 USB Mouse Driver Initialization

Locating This Component

Component Name
USB Mouse Driver Initialization

VxWorks Component Name
Mouse Init

VxWorks Component Tree Location
hardware > peripherals > USB Devices > USB Device Init > Mouse Init

Include Macro Name
#define INCLUDE_USB_MOUSE_INIT

Configuration Summary

Init Routine
usrUsbMseInit()

Configlettes
usrUsbMseInit.c

Dependencies

Requires INCLUDE_USB_MOUSE and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.13 USB Printer Driver

Locating This Component

Component Name
USB Printer Driver

VxWorks Component Name
Printer
A Runtime Configuration Summary
A.2 Wind River USB Host Stack Components

VxWorks Component Tree Location
   hardware > peripherals > USB Devices > Printer

Include Macro Name
   #define INCLUDE_USB_PRINTER

Libraries
   usbPrinterLib

Dependencies
   Requires INCLUDE_USB.

A.2.14 USB Printer Driver Initialization

Locating This Component

   Component Name
   USB Printer Driver Initialization

   VxWorks Component Name
   Printer Init

   VxWorks Component Tree Location
   hardware > peripherals > USB Devices > USB Device Init > Printer Init

   Include Macro Name
   #define INCLUDE_USB_PRINTER_INIT

Configuration Summary

   Init Routine
   usrUsbPrnInit( )

   Configlettes
   usrUsbPrnInit.c
Dependencies

Requires INCLUDE_USB_PRINTER. and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.15 **USB Speaker Driver**

Locating This Component

Component Name
USB Speaker Driver

VxWorks Component Name
Speaker

VxWorks Component Tree Location
hardware > peripherals > USB Devices > Speaker

Include Macro Name
#define INCLUDE_USB_SPEAKER

Libraries

usbSpeakerLib

Dependencies

Requires INCLUDE_USB.

A.2.16 **USB Speaker Driver Initialization**

Locating This Component

Component Name
USB Speaker Driver Initialization
VxWorks Component Name  
Speaker Init

VxWorks Component Tree Location  
hardware > peripherals > USB Devices > USB Device Init > Speaker Init

Include Macro Name  
#define INCLUDE_USB_SPEAKER_INIT

Configuration Summary

Init Routine  
usrUsbSpkrInit()

Configlettes  
usrUsbSpkrInit.c

Dependencies

Requires INCLUDE_USB_SPEAKER and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.

A.2.17 Bulk Only Mass Storage USB Driver

Locating This Component

Component Name  
Bulk Only Mass Storage USB Driver

VxWorks Component Name  
Mass Storage - Bulk

VxWorks Component Tree Location  
hardware > peripherals > USB Devices > Mass Storage - Bulk

Include Macro Name  
#define INCLUDE_USB_MS_BULKONLY
Libraries

usbBulkDevLib

Dependencies

Requires INCLUDE_USB.

A.2.18 Bulk Only Mass Storage USB Driver Initialization

Locating This Component

Component Name
Bulk Only Mass Storage USB Driver Initialization

VxWorks Component Name
Bulk Mass Storage Init

VxWorks Component Tree Location
hardware > peripherals > USB Devices > USB Device Init > Bulk Mass Storage Init

Include Macro Name
#define INCLUDE_USB_MS_BULKONLY_INIT

Configuration Summary

Init Routine
usrUsbBulkDevInit()

Configlettes
usrUsbBulkDevInit.c

Parameters

BULK_DRIVE_NAME
The USB Bulk Drive Name specifies the drive name assigned to the USB bulk only device. The default is “/bd”.

186
A.2 Wind River USB Host Stack Components

BULK_MAX_DEVS
The USB Bulk Maximum Drives specifies the maximum number of bulk device drives supported. The default is 2.

BULK_MAX_DRV_NAME_SZ
The USB Bulk Device Name Size specifies the maximum size of the USB bulk device name.

USB_BULK_NON_REMOVABLE_DISK
The USB Bulk Non-Removable Disk, when set to TRUE, specifies that the status check on the mass storage disk to determine whether the media is present or not is not performed to improve performance. The default is FALSE.

Dependencies
Requires INCLUDE_USB_MS_BULKONLY and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL. There is a compiler macro, BULK_RESET_NOT_SUPPORTED, that can be used to turn off mass storage reset. See Configuring Bulk Mass Storage Device Names, p.13.

A.2.19 USB Mass Storage Driver - Control Bulk Interrupt

Locating This Component

Component Name
Mass Storage USB Driver - Control/Bulk/Interrupt

VxWorks Component Name
Mass Storage - CBI

VxWorks Component Tree Location
hardware > peripherals > USB Devices > Mass Storage - CBI

Include Macro Name
#define INCLUDE_USB_MS_CBI

Libraries

usbCbiUfiDevLib
Dependencies

Requires INCLUDE_USB.

A.2.20 USB Mass Storage - CBI Class Driver Initialization

Locating This Component

Component Name
USB Mass Storage - CBI Class Driver Initialization

VxWorks Component Name
CBI Mass Storage Init

VxWorks Component Tree Location
hardware > peripherals > USB Devices > USB Device Init > CBI Mass Storage Init

Include Macro Name
#define INCLUDE_USB_MS_CBI_INIT

Configuration Summary

Init Routine
usrUsbCbiUfiDevInit()

Configlettes
usrUsbCbiUfiDevInit.c

Parameters

CBI_DRIVE_NAME
The USB CBI Drive Name, which specifies the name of the drive assigned to a
the USB CBI device. The default is “/cbid.”

UFI_MAX_DEVS
The Maximum UFI Devices, which specifies the maximum number of CBI
devices supported. The default is 2.
A.2.21 END - Pegasus USB Driver

Locating This Component

Component Name
END - Pegasus USB Driver

VxWorks Component Name
END - Pegasus

VxWorks Component Tree Location
hardware > peripherals > USB Devices > END - Pegasus

Include Macro Name
#define INCLUDE_USB_PEGASUS_END

Dependencies

Requires INCLUDE_USB.

A.2.22 END - Pegasus USB IPv4 Driver Initialization

Locating This Component

Component Name
End - Pegasus USB IPv4 Driver Initialization

VxWorks Component Name
End - Pegasus IPv4 Initialization
VxWorks Component Tree Location
  hardware > peripherals > USB Devices > USB Device Init >
  End - Pegasus IPv4 Initialization

Include Macro Name
#define INCLUDE_USB_PEGASUS_END_INIT

Configuration Summary

Init Routine
usrUsbPegasusEndInit()

Configlettes
usrUsbPegasusEndInit.c

Parameters

PEGASUS_IP_ADDRESS
The Pegasus IP Address, which specifies the IP address of the USB Pegasus device. The default is {"90.0.0.3"}.

PEGASUS_NET_MASK
The Pegasus Net Mask, which specifies the USB Pegasus Device Net Mask. The default is {0xffffff00}.

PEGASUS_TARGET_NAME
The Pegasus Target Name, which specifies the target name of the USB Pegasus device. The default is {"usbTarg0"}.

PEGASUS_MAX_DEVS
The USB Pegasus Device Maximum Number, which specifies the maximum number of supported Pegasus devices. The default is 1.

Dependencies

Requires INCLUDE_USB_PEGASUS_END and INCLUDE_USB_INIT. Should not be used with the USB Tool, INCLUDE_USBTOOL.
A.2.23 USB Audio Demo

Locating This Component

Component Name
USB Audio Demo for Host Stack

VxWorks Component Name
USB Audio Demo

VxWorks Component Tree Location
hardware > buses > USB Hosts > USB Host Init > USB Audio Demo

Include Macro Name
#define INCLUDE_USB_AUDIO_DEMO

Configuration Summary

Init Routine
usrUsbAudioDemo()

Configlettes
usrUsbAudioDemo.c

Dependencies

Requires INCLUDE_USB and INCLUDE_USB_SPEAKER. Should not be used with the USB Tool, INCLUDE_USBTOOL, or with INCLUDE_USB_SPEAKER_INIT (because the Speaker Audio Demo itself calls the speaker initialization routine).
A.3 Wind River USB Peripheral Stack Components

This section lists the components used for the Wind River USB peripheral stack.

A.3.1 USB Peripheral Stack

Locating This Component

<table>
<thead>
<tr>
<th>Component Name</th>
<th>VxWorks Component Name</th>
<th>VxWorks Component Tree Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB Peripheral Stack</td>
<td>USB Peripheral Stack</td>
<td>hardware &gt; buses &gt; USB Target &gt; USB Peripheral Stack &gt; USB Peripheral Stack</td>
</tr>
</tbody>
</table>

Include Macro Name

#define INCLUDE_USB_TARG

Configuration Summary

Init Routine

usrUsbTargPciInit()

Configlettes

usrUsbTargPciInit.c

A.3.2 NetChip NET2280 USB Peripheral Controller Driver

Locating This Component

Component Name

NetChip NET2280 USB Peripheral Controller Driver
A Runtime Configuration Summary
A.3 Wind River USB Peripheral Stack Components

VxWorks Component Name
NetChip NET2280

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Controllers > NetChip NET2280

Include Macro Name
#define INCLUDE_NET2280

Dependencies
Requires INCLUDE_USB_TARG.

A.3.3 Philips ISP 1582 USB Peripheral Controller Driver

Locating This Component

Component Name
Philips ISP1582 USB Peripheral Controller Driver

VxWorks Component Name
PhilipsIsp1582

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Controllers > PhilipsIsp1582

Include Macro Name
#define INCLUDE_PHILIPS1582

Configuration Summary

Init Routine
sysIsp1582PciDisconnect()

Dependencies
Requires INCLUDE_USB_TARG.
A.3.4 PDIUSBD12 USB Peripheral Controller Driver

Locating This Component

Component Name
PDIUSBD12 USB Peripheral Controller Driver

VxWorks Component Name
PDIUSBD12

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Controllers > PDIUSBD12

Include Macro Name
#define INCLUDE_PDIUSBD12

Dependencies

Requires INCLUDE_USB_TARG.

A.3.5 USB Keyboard Emulator

Locating This Component

Component Name
USB Keyboard Emulator

VxWorks Component Name
Keyboard Emulator

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Stack > Keyboard Emulator

Include Macro Name
#define INCLUDE_KBD_EMULATOR
A Runtime Configuration Summary
A.3 Wind River USB Peripheral Stack Components

Libraries

usbTargKbdLib

Dependencies

Requires INCLUDE_USB_TARG.

A.3.6 Target Keyboard Emulator Driver Initialization

Locating This Component

Component Name
Targ Keyboard Emulator Driver Initialization

VxWorks Component Name
Targ Keyboard Emulator Init

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Init > Targ Keyboard Emulator Init

Include Macro Name
#define INCLUDE_KBD_EMULATOR_INIT

Configuration Summary

Init Routine
usrUsbTargKbdInit()

Configlettes
usrUsbTargKbdInit.c

Dependencies

Requires INCLUDE_KBD_EMULATOR. The USB Peripheral Keyboard function driver initialization should not be used with the USB Tool, INCLUDE_USBTOOL.
A.3.7 USB Mass Storage Emulator

Locating This Component

Component Name
USB Mass Storage Emulator

VxWorks Component Name
Mass Storage Emulator

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Stack > Mass Storage Emulator

Include Macro Name
#define INCLUDE_MS_EMULATOR

Libraries

usbTargMsLib       usbTargRbcLib

Dependencies

Requires INCLUDE_USB_TARG.
The USB Peripheral Mass Storage function driver requires the dosFs file system components. See 2. Configuring VxWorks Components.

A.3.8 Target Mass Storage Emulator Driver Initialization

Locating This Component

Component Name
Targ Mass Storage Emulator Driver Initialization

VxWorks Component Name
Targ Mass Storage Emulator Init
A.3 Wind River USB Peripheral Stack Components

VxWorks Component Tree Location
  hardware > buses > USB Target > USB Peripheral Init >
  Targ Mass Storage Emulator Init

Include Macro Name
  #define INCLUDE_MS_EMULATOR_INIT

Configuration Summary

  Init Routine
    usbTargMsInit()

  Configlettes
    usrUsbTargMsInit.c

Dependencies

  Requires INCLUDE_MS_EMULATOR. The USB Peripheral Mass Storage function
  driver initialization should not be used with the USB Tool, INCLUDE_USBTOOL.

A.3.9 USB Printer Emulator

Locating This Component

  Component Name
    USB Printer Emulator

  VxWorks Component Name
    Printer Emulator

  VxWorks Component Tree Location
    hardware > buses > USB Target > USB Peripheral Stack > Printer Emulator

  Include Macro Name
    #define INCLUDE_PRN_EMULATOR
Libraries

usbTargPrnLib

Dependencies

Requires INCLUDE_USB_TARG.

A.3.10 Target Printer Emulator Driver Initialization

Locating This Component

Component Name
Targ Printer Emulator Driver Initialization

VxWorks Component Name
Targ Printer Emulator Init

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Init >
Targ Printer Emulator Init

Include Macro Name
#define INCLUDE_PRN_EMULATOR_INIT

Configuration Summary

Init Routine
usrUsbTargPrnInit()

Configlettes
usrUsbTargPrnInit.c

Dependencies

Requires INCLUDE_PRN_EMULATOR. The USB Peripheral Printer function driver initialization should not be used with the USB Tool, INCLUDE_USBTOOL.
A.3.11 USB D12 Emulator

Locating This Component

Component Name
USB D12 Emulator

VxWorks Component Name
D12 Emulator

VxWorks Component Tree Location
hardware > buses > USB Target > USB Peripheral Stack > D12 Emulator

Include Macro Name
#define INCLUDE_D12_EMULATOR

Libraries

usbTargPhilipsD12EvalLib

Dependencies

Requires INCLUDE_USB_TARG.

A.4 USB Tool

Locating This Component

Component Name
USB Tool - Test Application for Host Stack

VxWorks Component Name
usbTool
VxWorks Component Tree Location

hardware > buses > usbTool

Include Macro Name
#define INCLUDE_USBTOOL

Configuration Summary

Configlettes

usrUsbTool.c

Dependencies

The USB Tool should not be used with any of the USB emulator driver initialization (INCLUDE_foo_EMULATOR_INIT) configurations.

Should not be used with any USB initialization components (INCLUDE Xxx_INIT) as listed below:

- the host stack initialization component
  INCLUDE_USB_INIT
- the host controller interface initialization components
  INCLUDE_EHCI_INIT, INCLUDE_OHCI_INIT, and INCLUDE_UHCI_INIT
- the USB driver initialization components
  INCLUDE_USB_MOUSE_INIT, INCLUDE_USB_KEYBOARD_INIT,
  INCLUDE_USB_PRINTER_INIT, INCLUDE_USB_SPEAKER_INIT, and
  INCLUDE_USB_PEGASUS_END_INIT
- the mass storage initialization components
  INCLUDE_USB_MS_BULKONLY_INIT and INCLUDE_USB_MS_CBI_INIT
- the speaker demo (which excludes ubsTool)
  INCLUDE_USB_AUDIO_DEMO
B.1 Glossary Terms

active device

An active device is a device that is powered and is not in the suspend state.

attach

To attach the target application and the TCD together, so that the target application can receive and respond to requests from the host through that TCD, the application in the configlette uses a target layer routine.

audio device

An audio device is a device that sources or sinks sampled analog data.

bandwidth

The bandwidth is the amount of data transmitted per unit of time, typically bits per second (b/s) or bytes per second (B/s).
big endian

A big endian is a method of storing data that places the most significant byte of multiple-byte values at a lower storage address. For example, a 16-bit integer stored in big endian format places the least significant byte at the higher address and the most significant byte at the lower address. See also little endian.

board support package

A board support package (BSP) consists of C and assembly source files which configure the VxWorks kernel for the specific hardware on your target board. The BSP-specific file for USB is /target/config/BSP/usbPciStub.c.

build flag

A build flag variable is defined in makefile and used to set various compiler options.

build macro

A build macro is a build flag, p.202.

bulk transfer

Bulk transfer is one of the four USB transfers. These are non periodic, large bursty communication typically used for a transfer that can use any available bandwidth and can also be delayed until bandwidth is available.

bus reset

A bus reset is a first signal sent by the host to the device once the host has detected a connect to the device on the bus. The control pipes should be created on a bus reset. See also, control pipe, p.203.

callback routine

A callback routine is registered with the lower layers by the client module. Callback routines are mapped to events, so that when a specific event occurs, the lower layers call the corresponding routine to signal the client module that the event has occurred.

channel

A channel is a communication path between the target application and target layer.
client

The client is software resident on the host that interacts with the USB system software to arrange data transfer between a function and the host. The client is often the data provider and consumer for transferred data.

compile-time macro

A compile-time macro is a build flag.

configlette

A configlette is a vxWorks configuration routine that initializes the USB components. All configlette routines are located in the directory, /target/config/comps/src.

control endpoint

A control endpoint is an IN/OUT device endpoint pair used by a control pipe. Control endpoints have the same endpoint number and transfer data in both directions; therefore, they use both endpoint directions of a device address and endpoint number combination. Thus, each control endpoint consumes two endpoint addresses.

control pipe

A control pipe is a bi-directional pipe that transfers data to the control endpoint. The data has an imposed structure that allows requests to be reliably identified and communicated.

control transfer

A control transfer is one of the four USB transfers. It supports configuration/command/status type communication between the client and function.

default address

A default address is an address defined by the USB Specification and used by a USB device when it is first powered or reset. The default address is 00H.

default control pipe

A default control pipe provides a message pipe through which a USB device can be configured once it is attached and powered. It is used for control transfers. All other pipes come into existence when the device is configured.
descriptor

A descriptor is a data structure with a defined format. A USB device reports its attributes to the host using descriptors. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor, followed by a byte-wide field that identifies the descriptor type.

detach

To detach is to remove the attach of the target application with the target controller driver.

device

A device is a logical or physical entity that performs a function. The actual entity described depends on the context of the reference. At the lowest level, a device may refer to a single hardware component, as in a memory device. At a higher level, it may refer to a collection of hardware components that perform a particular function, such as a USB interface device. At an even higher level, device may refer to the function performed by an entity attached to the USB; for example, a data/FAX modem device. The term device is sometimes interchanged with peripheral.

device address

A device address is a value between 0 and 127, inclusive that the target application uses to identify the device in the bus.

device qualifier

A device qualifier descriptor describes information about a high-speed capable device that would change if the device were operating at the other speed.

disable

To disable the USB peripheral stack is to make it invisible to the USB host. Once disabled, the peripheral stack does not respond to any request from the host.

disconnect

To disconnect is to unplug the USB device connection from the host.

downstream

Downstream indicates the direction of data flow from the host or away from the host. A downstream port is the port on a hub electrically farthest from the host that
generates downstream data traffic from the hub. Downstream ports receive upstream data traffic.

driver

When referring to hardware, a driver is an I/O pad that drives an external load. When referring to software, a driver is a program responsible for interfacing to a hardware device, that is, a device driver.

Enhanced Host Controller Interface

The Enhanced Host Controller Interface (EHCI) is the host controller compliant to USB 2.0 specification.

enable

To enable a USB device is to bring it up in such a way that it is visible to and able to respond to a USB host.

endpoint

An endpoint is one of a number of enumerable sources or destinations for data on a USB-capable peripheral or host.

endpoint descriptor

An endpoint descriptor is a data structure that describes an endpoint (see Endpoint Descriptor, p.128).

endpoint number

An endpoint number is a four-bit value between 0H and FH, inclusive, associated with an endpoint on a USB device.

endpoint request packet

An endpoint request packet (ERP) is a structure that is defined in target/h/usb/usb.h. It consists of data that the peripheral wishes to communicate to the host.

enumerate

To enumerate a device is to get it detected and configured as a USB device on the host. Once the device is detected, the host sends a bus reset, followed by a sequence of standard requests that the device should respond to, in order to get it enumerated (configured).
feature

A feature is the element to be set or cleared for a GET_FEATURE or CLEAR_FEATURE request from host. See Feature Parameter, p.118.

first in, first out

A first in, first out (FIFO) protocol is an approach to handling program work requests from a queue or stack so that the request that is first to enter the queue, is first to be addressed.

frame

A frame is a 1-millisecond time base, established on full-/low-speed buses.

frame number

A frame number is an identifier to the active frame.

full-speed

Full-speed refers to USB operation at 12 Mb/s. See also low-speed and high-speed.

function driver

A function driver implements the functionality of a device, such as a mass storage device.

generic endpoints

The generic endpoints are all the endpoints used to support bulk, isochronous, and interrupt transfers. Generic endpoints do not include control endpoints (see control endpoint, p.203).

halt


handle

A handle is an unique identifier that used for communication with the object to which it is assigned. For example, the target controller driver provides the target application with a handle during the attachment process. The target application uses this handle to communicate with the attached TCD.

As with TCDs, there are handles to every pipe created by the target application. The target application uses this handle to communicate with the pipe.
**hardware adaptation layer**

The *hardware adaptation layer* (HAL) provides a hardware-independent view of the target controller to higher layers in the stack.

**high-speed**

*High-speed* refers to USB operation at 480 Mb/s. See also *low-speed* and *full-speed*.

**host**

The *host* computer system is where the USB host controller is installed. This includes the host hardware platform (such as the CPU, bus, and so on) and the operating system in use.

**host controller**

The *host controller* is the host's USB interface.

**host controller driver**

The *host controller driver* is the USB software layer that abstracts the host controller hardware. The HCD provides an SPI for interaction with the host controller. The host controller driver hides the specifics of the host controller hardware implementation from the above layers of the stack.

**host stack**

The USB *host stack* is software that enables a function driver to communicate with the USB device.

**hub**

A *hub* is a USB device that provides additional connections to the USB.

**I/O request packet**

An *I/O request packet* (IRP) is an identifiable request by a software client to move data between itself (on the host) and an endpoint of a device in an appropriate direction.

**IN endpoint**

The *IN endpoint* is the endpoint that corresponds to IN requests from host.
interrupt endpoint

An interrupt endpoint is the endpoint associated with interrupt-type transfers.

interrupt request

An interrupt request (IRQ) is a hardware signal that allows a device to request attention from a host. The host typically invokes an interrupt service routine to handle the condition that caused the request.

interrupt request line

An interrupt request line is a hardware line over which devices can send interrupt signals to the microprocessor.

interrupt transfer

An interrupt transfer is one of the four USB transfers. It is characterized by small data, non-periodic, low frequency and bounded latency. It is typically used to handle service needs.

isochronous transfer

An isochronous transfer is one of the four USB transfer types. Isochronous transfers are used when working with isochronous data. Isochronous transfers provide periodic, continuous communication between host and device.

little endian

Little endian is a method of storing data that places the least significant byte of multiple-byte values at lower storage addresses. For example, a 16-bit integer stored in little endian format places the least significant byte at the lower address and the most significant byte at the next address. See also big endian.

low-speed

Low-speed is USB operation at 1.5 Mb/s. See also full-speed and high-speed.

message pipe

A message pipe is a bi-directional pipe that transfers data using a request/data/status paradigm. The data has an imposed structure that allows requests to be reliably identified and communicated.
microframe

A microframe is a 125-microsecond time base established on high-speed buses.

Open Host Controller Interface

The Open Host Controller Interface (OHCI) is the host controller compliant to the USB 1.1 specification.

OUT endpoint

The OUT endpoint is the endpoint that corresponds to OUT requests from host.

peripheral

See device, p.204.

peripheral stack

The peripheral stack is the software on the USB peripheral that interprets and responds to the commands sent by the USB host.

pipe

A pipe is a logical abstraction representing the association between an endpoint on a device and the software on the host. A pipe has several attributes; for example, a pipe may transfer data as streams (stream pipe) or as messages (message pipe).

pipe handle

A pipe handle is used by the target application when it carries out USB transfer on an endpoint.

In order to abstract the pipe information which is maintained internally by the peripheral stack from the target application, only a handle (which is a number which identifies the pipe) is given to the target application. This is just an identifier for the pipe, and does not represent the endpoint number.

polling

Polling is the method used to ask multiple devices, one at a time, if they have any data to transmit.

protocol

A protocol is a specific set of rules, procedures, or conventions relating to format and timing of data transmission between two devices.
release

The release is the USB (release) number obtained from the device descriptor.

remote wakeup

A remote wakeup is an event generated by a device to bring the host system out of a suspended state. See also, resume, p.210.

reset


resume

A resume is a signal sent by the host to make the device that is in a suspended state come out of the suspended state and restart. See also, remote wakeup, p.210.

root hub

A root hub is the USB hub directly attached to the host controller. This hub (tier 1) is attached to the host.

root port

The root port is the downstream port on a root hub.

setup

The setup is the first transaction initiated from the host to a device during a control transfer.

setup packet

The setup packet contains a USB-defined structure that accommodates the minimum set of commands required to enable communication between the host and a device.

stall

The target application may stall a generic endpoint in response to certain error conditions, which will indicate to the host that an error has occurred on the target.
standard request

A standard request is a certain USB request, such as GET_STATUS or SET_CONFIGURATION, which all USB devices should support. Devices should respond to this request even if it is not assigned an address or configured.

start-of-frame

The start-of-frame (SOF) is the first transaction in each (micro)frame. An SOF allows endpoints to identify the start of the (micro)frame and synchronize internal endpoint clocks to the host.

state

A state is the last known status or condition of a process, application, object, or device. The state of an endpoint indicates whether or not it is stalled.

status

See state.

suspend

To suspend is to put into a non-active state in order to conserve power. For example, the host may suspend a device, for a specific period of time, when it observes that there is no traffic on the bus. When suspended, the USB device maintains any internal status, including its address and configuration.

target application

A target application responds to USB requests from the host that the TCD routes to the target application through the target layer.

target channel

See handle, p.206.

target controller

A target controller (TC) is the hardware part of the peripheral that connects to the USB.

target controller driver

The target controller driver (TCD) is the driver that sits just above target controller and carries out all of the hardware-specific implementation of the target controller.
target layer

The target layer is a consistent, abstract mediator between a variety of target applications and the hardware adaptation layer.

target request block

The target request block (TRB) is a request block created by HAL. It consists of the handle to the TCD, function code, the length of the TRB, and the parameter associated with the function code. Corresponding to every function code, there is an associated TRB. The HAL passes this request block to the TCD through the single entry point.

test mode

A test mode is a state used for debugging purposes. To facilitate compliance testing, host controllers, hubs, and high-speed capable functions must support various test modes as defined in the USB 2.0 Specification.

token packet

A token packet is a type of packet that identifies what transaction is to be performed on the bus.

toolchain

The toolchain is the name of the compiler used to build the source files. The toolchain is specified in the make command with the argument TOOL.

transaction

The transaction is the delivery of service to an endpoint. It consists of a token packet, optional data packet, and optional handshake packet. Specific packets are allowed/required based on the transaction type.

transfer type

The transfer type determines the characteristics of the data flow between a software client and its function. Four standard transfer types are defined: control transfer, interrupt transfer, bulk transfer, and isochronous transfer.

translation unit

A translation unit is a thin layer of the software that provides the backward compatibility between the USB 2.0 host stack and the USB 1.1 class drivers. For details, refer to 3.2 Architecture Overview, p.31.
Universal Host Controller Interface

The Universal Host Controller Interface (UHCI) is the host controller compliant to USB 1.1 specification.

Universal Serial Bus Driver

The Universal Serial Bus Driver (USBD) is the host resident software entity responsible for providing common services to clients that are manipulating one or more functions on one or more host controller. It is a hardware independent software layer that implements the USB protocol 2.0. It acts as a channel between the class drivers and host controller driver. For details, refer to 3.2 Architecture Overview, p.31.

USB Request Block

The USB Request Block (URB) is used to send or receive data to or from a specific USB endpoint on a specific USB device in an asynchronous manner.

USB Device

A USB device is a hardware device that performs a useful end-user function. Interactions with USB devices flow from the applications through the software and hardware layers to the USB devices.

wakeup

A wakeup is an event through which the device comes out of a suspended state.
## B.2 Abbreviations and Acronyms

The Wind River USB documentation uses the following abbreviations and acronyms:

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP</td>
<td>Board Support Package</td>
</tr>
<tr>
<td>CBI</td>
<td>Command-Bulk-Interrupt</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>EHCI</td>
<td>Enhanced Host Controller Interface</td>
</tr>
<tr>
<td>END</td>
<td>Enhanced Network Driver</td>
</tr>
<tr>
<td>ERF</td>
<td>Event Reporting Framework</td>
</tr>
<tr>
<td>ERP</td>
<td>Endpoint Request Packet</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Adaptation Layer</td>
</tr>
<tr>
<td>HC</td>
<td>Host Controller</td>
</tr>
<tr>
<td>HCD</td>
<td>Host Controller Driver</td>
</tr>
<tr>
<td>HID</td>
<td>Human Interface Device</td>
</tr>
<tr>
<td>HRFS</td>
<td>Highly Reliable File System</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IRP</td>
<td>I/O Request Packet</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt Request</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Pictures Experts Group</td>
</tr>
<tr>
<td>OHCI</td>
<td>Open Host Controller Interface</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Control Interface</td>
</tr>
<tr>
<td>SIO</td>
<td>Serial I/O</td>
</tr>
</tbody>
</table>
### Abbreviations and Acronyms (cont'd)

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOF</td>
<td>start-of-frame</td>
</tr>
<tr>
<td>SPC-2</td>
<td>SCSI Primary Commands - 2</td>
</tr>
<tr>
<td>TC</td>
<td>Target Controller</td>
</tr>
<tr>
<td>TCD</td>
<td>Target Controller Driver</td>
</tr>
<tr>
<td>TRB</td>
<td>Target Request Block</td>
</tr>
<tr>
<td>UHCI</td>
<td>Universal Host Controller Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>USBD</td>
<td>Universal Serial Bus Driver</td>
</tr>
<tr>
<td>XBD</td>
<td>Extended Block Device</td>
</tr>
</tbody>
</table>
C

usbTool Code Exerciser Utility Tool

C.1 Introduction

Wind River USB includes a utility called **usbTool** that you can use to exercise the modules that compose Wind River USB. For example, if you are implementing a new device driver, you can use **usbTool** to exercise and debug your code. The **usbTool** utility also provides a basic code skeleton that you can customize to suit your test needs.

This chapter describes the USB utility tool. For additional information on configuring VxWorks to include this component, see *Locate and include the USB Testing Tool*, p.13 and *Adding the USB Tool Code Exerciser Component*, p.21.

**NOTE:** The **usbTool** utility relies internally on static data. Do not run multiple instances of **usbTool** simultaneously.
C.2 Running usbTool from the Shell

1. To use `usbTool` from the shell, type the entry point of the tool at the shell prompt, as follows:

   ```
   ->usbTool
   ```

   This produces a new prompt where commands can be entered:

   ```
   usb>
   ```

2. To get a list of all `usbTool` commands and their definitions, type the following:

   ```
   usb>?  
   ```

   **CAUTION:** When running `usbTool` on a host shell, you may encounter a redirection problem that prevents entered commands from being echoed back to the shell. You can avoid this issue by running `usbTool` on a target shell.

C.3 Using the usbTool Execution Sequence

The `usbTool` utility allows the user to see the typical execution sequence needed to get the USB up and running.

When an image that includes USB components boots, it automatically executes a sequence of events similar to using the `usbTool` utility:

1. To initialize the USB host stack, enter the `usbInit` command at the `usbTool` prompt.

2. To initialize an EHCI, OHCI, or UHCI host controller, enter the `attach ehci`, `attach ohci`, or `attach uhci` commands at the `usbTool` prompt.

3. If you plan to use a device driver, after initializing the host stack and host controller, enter the initialization command for that driver at the `usbTool` prompt.

   To test any included devices, enter the appropriate test command, for example:

   ```
   usb>mouseTest
   ```
The following commands are available when their associated device components are included in your VxWorks image:

- mouse
  - mouseTest
- keyboard
  - kbdInit, kbdDown, and kbdPoll
- printer
  - prnInit, prnDown, print4k, and print filename
- speaker
  - sblkInit, sblkDown, sblkFmt, volume, and play filename

For descriptions of these commands, invoke the help list from usbTool.

**NOTE:** Command names do not have to be fully entered to be executed. For example, usbInit can be entered as usbi.

### C.4 Testing Applications

This section describes how to test applications using the usbTool utility.

#### C.4.1 Testing the Keyboard Application

To test the keyboard test application, proceed as follows.

**Step 1:** Start the keyboard test application.

To start the keyboard test application, invoke the usbTool on the target, and then from its prompt, give the `targInit TCD kbd` command, where TCD can be `d21`, `isp1582`, or `net22280`. The following example shows this command and the resulting output for a PDIUSB12 controller:
Step 2: Connect the target with the host and invoke `usbTool`.

Next, with a USB cable connect the target containing the peripheral stack with keyboard functionality to the USB host stack. Invoke the `usbTool` application on the host by using the `usbTool` command.

Step 3: Initialize the host and attach the controller.

Initialize the USB host stack with the `usbInit` command. Attach the host controller with the command `attach x HCI` (where x is “o”, “u”, or “e”).

Step 4: Issue the `usbEnum` command.

On the host, at the `usbTool` prompt give the `usbEnum` command. The output should resemble the following:

```bash
usb> usbEnum
bus count = 1
    enumerating bus 0
        hub 0x1
            port count = 5
            port 0 not connected
            port 1 not connected
            port 2 is hub 0x2
                hub 0x2 = NEC Corporation/USB2.0 Hub Controller
                    port count = 4
                    port 0 not connected
                    port 1 not connected
                    port 2 not connected
                    port 3 is device 0x3 = Wind River Systems/USB keyboard emulator
                        port 3 not connected
                        port 4 not connected
```

Step 5: Initialize the keyboard driver.

Initialize the keyboard driver on the host by using the `kdbInit` command on the `usbTool` prompt.

Step 6: Poll the driver.

Then, from the `usbTool` prompt at the host, give the `kbdPoll` command to poll the Keyboard SIO Driver for input. Then, from the target shell, give the `kbdReport` command (if no parameters are given to `kbdReport`, it will by default send the characters “a” through “z” from the device to the host).
The host console should display the following output:

```
usb> kbdPoll
ASCII  97 'a'
ASCII  98 'b'
ASCII  99 'c'
ASCII 100 'd'
ASCII 101 'e'
ASCII 102 'f'
ASCII 103 'g'
ASCII 104 'h'
ASCII 105 'i'
ASCII 106 'j'
ASCII 107 'k'
ASCII 108 'l'
ASCII 109 'm'
ASCII 110 'n'
ASCII 111 'o'
ASCII 112 'p'
ASCII 113 'q'
ASCII 114 'r'
ASCII 115 's'
ASCII 116 't'
ASCII 117 'u'
ASCII 118 'v'
ASCII 119 'w'
ASCII 120 'x'
ASCII 121 'y'
ASCII 122 'z'
ASCII  26
Stopped by CTRL-Z
```

C.4.2 Testing the Printer Application

To test the printer application, proceed as follows.

**Step 1:** Start the printer test application.

To start the printer test application, invoke the `usbTool` on the target and then from its prompt, give the `targInit TCD prn` command, where TCD can be `d21`, `isp1582`, or `net22280`. The following example shows this command and the resulting output for a PDIUSBD12 controller:

```
usb> targInit d12 prn
usbTargInitialize() returned OK
Philips PDIUSBD12: ioBase = 0x368, irq = 7, dma = 3
usbTargTcdAttach() returned OK
targChannel = 0x1
usbTargEnable() returned OK
```
Step 2: Connect the target with the host.

Next, with a USB cable connect the target containing the peripheral stack with printer functionality to the USB host stack.

Step 3: Initialize the host and attach the controller.

Initialize the USB host stack with the `usbi` command and attach the host controller with the command `attach xhci` (where x is “o”, “u”, or “e”).

Step 4: Issue the `usbEnum` command.

On the host, at the `usbTool` prompt give the `usbEnum` command. The output should resemble the following:

```
usb> usbEnum
bus count = 1
   enumerating bus 0
      hub 0x1
         port count = 5
         port 0 not connected
         port 1 not connected
         port 2 is hub 0x2
            hub 0x2 = NEC Corporation/USB2.0 Hub Controller
               port count = 4
                   port 0 not connected
                   port 1 is device 0x3 = Wind River Systems/USB printer emulator
                   port 2 not connected
                   port 3 not connected
                  port 4 not connected
```

Step 5: Initialize the printer driver.

Initialize the printer driver on the host by giving the `prnInit` command on the `usbTool` prompt. Then print 4K blocks of data, by giving the following command from the host (where n can be 1, 2, 3, etc.):

```
usb> print4k n
```

Step 6: Issue the `prnDump` command.

From the target peripheral, give the `prnDump` command. This results in output on the target, like the output shown below:

```
usb> prnDump
Printer received 4096 bytes.
00 01 00 03 00 05 00 07 00 09 00 0b 00 0d 00 0f .................
00 11 00 13 00 15 00 17 00 19 00 1b 00 1d 00 1f ...................,
00 21 00 23 00 25 00 27 00 29 00 2b 00 2d 00 2f .!.#%`.),+-./
00 31 00 33 00 35 00 37 00 39 00 3b 00 3d 00 3f .1.3.5.7.9.;=?,
00 41 00 43 00 45 00 47 00 49 00 4b 00 4d 00 4f .A.C.E.G.I.K.M.O
00 51 00 53 00 55 00 57 00 59 00 5b 00 5d 00 5f .Q.S.U.W.Y.[]_,
```
and in output on the host that looks like the output shown below:

```
usb> print4k 1
Device ID length = 25
Device ID = mfg:WRS;model=emulator;protocol=0xff (unknown)
sending 1 4k blocks to printer...
usb>
```

You could also issue the command `print "filename"` from the host, and `prnDump` from the target. This prints the contents of the file rather than the 4k test data blocks. If there is nothing to print, the message "Printer has no data" will be displayed instead.

### C.4.3 Testing the Mass Storage Application

After peripheral mass storage function initialization, if the host has mass storage class driver initialization built-in it should recognize that a mass storage device is attached. You can then do normal file system operations on the USB disk, as in:

```c
    cd("/bd0")
    creat("temp.txt",2)
    rm("temp.txt")
```
Index

A

abbreviations 214
acronyms 214
actLen
  USB_BFR_LIST 134
  USB_ERP 135
address range translation routines
  PCI and non-PCI 171
addressed state
  interfaceGet() 123
  interfaceSet() 123
addressSet()
    callback response to SET_ADDRESS 125
    table of associated control requests 113
alternateSetting
  interfaceSet() 123
attach 201
attach ehci 218
attach ohci 218
attach uhci 218
attach xhci
  keyboard testing 220
  printer testing 222
attaching a TCD
  usbTargetEnable() 106
attributes
  USB_ENDPOINT_DESCR 128
audio device 201

AudioStreaming 91

B

bandwidth 201
bConfigurationValue 128
bfrCount
  USB_ERP 134
bfrLen
  USB_BFR_LIST 134
bfrList 132, 134
  bfrCount 134
  erpLen 132
  pid element 134
  USB_ERP 134
big endian 202
big-endian
  sample code for non-PCI OHCI controller 168
big-endian platforms
  host stack tested on 157
bit data I/O, routines for 171
Block_IO structure 96
bmRequestType 126
board support package, see BSP
boot parameter string 17
bRequest 126
BSP 6, 16, 168, 202
build flag 202
build macro 202
Bulk Device Name Size 187
Bulk Drive Name 186
Bulk Mass Storage Init 186
Bulk Maximum Drives 187
Bulk Non-Removable Disk 187
Bulk Only Mass Storage USB Driver component 185
Bulk Only Mass Storage USB Driver Initialization component 186
bulk transfer 202
bulk transfers 2
Bulk-Only driver 93
bus enumeration routines 52
bus reset 202
TARG_MNGMT_BUS_RESET management event code 113
bus tasks, overview 38

callback routine 202
callback table, target application 108
callbackInstaI( ) 83
callbackParam
usbTargTcdAttach( ) 105, 112
callbacks 40, 48
cancelURB( ) 62
CBI Drive Name 188
CBI driver 93
CBI Mass Storage Init 188
CBI_DRIVE_NAME 188
channel 202
CLEAR_FEATURE
clearing a device feature
hardware layer 147
target layer 137
featureClear( ) callback 116
setting and clearing device-specific features 147
table of control requests 112
usbHstClearFeature( ) in response to 137
client 203
client callback tasks 48
client modules 38, 47
cmdParser 34
code exerciser
usbTool 217
communication class device initialization 27
communication class drivers 99
data flow 102
dynamic attachment 101
END driver model 99
Ethernet networking control model driver 99
initialization 101
interrupt behavior 102
ioctl routines 102
compile-time macro 203
components
reference summary 173
config.h 16, 17
configAll.h 16, 17
configlette 203
configuration stage
addressSet( ) 125
configurationGet( ) 119
defined 119
table of associated control requests 112
configurationSet( )
defined 119
table of associated control requests 112
configuring
bulk mass storage device names
in config.h 18, 19
with the IDE 13
CBI mass storage device names
in config.h 18, 19
with the IDE 14
host controller driver 17
mass storage device names
in config.h 18, 19
with the IDE 14
network device parameters
in config.h 19
with the IDE 14
Pegasus network device parameters
in config.h 19
Index

with the IDE 14
USB class drivers 18, 19
USB components 16
in the project facility 10
USB hardware 158
USB host driver 17
usbttool 21
connecting a device to a hub 69
control data transfer 130
control endpoint 203
control pipe 203
custom transfer 203
custom transfers 2
createPipe() 61
d
D12 Emulator 199
data flow 56
data structures
  USBHST_URB 43
data transfer interfaces 42
data transfer rates 2
dataToggle
  USB_ERP 134
default address 203
default control pipe 203
default control pipes 5, 128
default state
  interfaceGet() 123
  interfaceSet() 123
  statusGet() 125
definitions 201
deletePipe() 61
demo applications
  usbAudio 27
descriptor 204
descriptor type
table of macros 121
descriptorGet() 120
defined 120
table of associated control requests 112
descriptorIndex
descriptorGet() 120
descriptorSet() 120
definition 120
table of associated control requests 112
descriptorType
descriptorGet() 120
descriptorSet() 120
parameter described 121
USB_ENDPOINT_DESCR 128
detach 204
device 204
device address 204
device configuration 52
device controller I/O addresses 17
device initialization 25
device qualifier 204
DeviceAdd_Callback 40
deviceAddress
  addressSet() 125
  TCD_FNC_ADDRESS_SET 147
  TRB_ADDRESS_SET 147
DeviceRemove_CallBack 40
DeviceResume_Callback 40
DeviceSuspend_Callback 40
disable 204
disconnect 204
disconnect event 154
disconnect signal 113
dma 107
dma element 107
downstream 204
driver 205
dynamic attachment registration 49
E
EHCI 175
host controller initialization 11
overview of host controllers 33
specification 2, 7
EHCI_Init 176
EHCI USB Host Controller Driver Initialization component 176
EHCI_PGMIF host controller value 162
enable 205
enabling and disabling the TCD 110
End - Pegasus IPv4 Initialization 189
End - Pegasus USB IPv4 Driver Initialization component 189
END driver
  USB Pegasus 22
END driver model 99
END- Pegasus 189
END- Pegasus USB Driver component 189
endian conversion
  non-PCI buses 169
  PCI buses 166
  USB 2.0 166
endpoint 205
endpoint attributes 128
endpoint descriptor 205
described 128
setting the 127
  USB_ENDPOINT_DESCR structure 128
endpoint interrupt status errors 153
endpoint number 205
endpoint request packet, see ERP
endpointAddress
  USB_ENDPOINT_DESCR 128
endpointId
  USB_ERP 134
endpoints
  number supported 146
Enhanced Host Controller Interface component 175
Enhanced Host Controller Interface, see EHCI
ENOSYS 78, 83
enumerate 205
ERF
device driver model
ERP
  completion result 132
type 134
  ERP_CALLBACK 133
callerCallback() 133
erpLen
  USB_ERP 132
error reporting 65

Ethernet networking control model driver 99
Event Reporting Framework, see ERF
Extended Block Device, see XBD

F

feature 206
  featureClear( ) 118
  featureSet( ) 118
feature selectors 118
featureClear( )
  table of associated control requests 112
featureSet( )
  table of associated control requests 112
FIFO
file system components 21
first in, first out, see FIFO
frame 206
  frame number 206
  retrieving 137
frameNo
  TRB_CURRENT_FRAME_GET 148
full data transfer rate 2
full-speed 206
function
  TRB header 143
function codes 144
function driver 206

G

generic data transfer 130
generic endpoints 206
GET_CONFIGURATION 41
  request, response to 119
table of control requests 112
GET_DESCRIPTOR 41
  descriptorGet( ) 120
table of control requests 112
GET_INTERFACE
  control request for interfaceGet( ) 113
**Index**

interfaceGet() callback 123
usbHstGetInterface() 41

GET_STATUS
getting the endpoint status 150
statusGet() callback 124
table of control requests 113
usbHstGetStatus() 42

GET_SYNCH_FRAME
table of control requests 113
getFrameNumber() 62
glossary 201

**H**

HAL 5, 142, 207
halt 206
handle 206
TRB header 143, 146
hardware adaptation layer, see HAL
HCD 3
HID_MSE_BOOT_REPORT 83
high data transfer rate 2
high-speed 207
host 207
host controller 207
initialization 11, 23
locating 162
overview of specifications 2
specification documentation 7
host controller driver 36, 207
configuring in config.h 17
EHCI 33
error reporting conventions 65
interfaces 61
OHCI 33
overview 59
registering a bus for the host controller 63
UHCI 33
host controller driver, see HCD 3
host stack 207
initialization 23
tested on big- and little-endian platforms 157
hub 207
hub class driver 35
hub class drivers 68
registering 68
hub functionality 33

I/O mapped PCI-based controller
bit data routines 171
I/O request packet, see IRP
IEEE-1284 87
IN endpoint 207
INCLUDE_AUDIO_DEMO 27
INCLUDE_D12_EMULATOR 199
INCLUDE_EHCI 11, 176
INCLUDE_EHCI_INIT 12, 176
INCLUDE_IFCONFIG 22
INCLUDE_KBD_EMULATOR 16, 194
INCLUDE_KBD_EMULATOR_INIT 16, 195
INCLUDE_MS_EMULATOR 16, 196
INCLUDE_MS_EMULATOR_INIT 16, 197
INCLUDE_NET2280 15, 193
INCLUDE_OHCI 11, 16, 177
INCLUDE_OHCI_INIT 12, 177
INCLUDE_PDIUSBD12 16, 194
INCLUDE_PHILIPS1582 16, 193
INCLUDE_PRN_EMULATOR 16, 197
INCLUDE_PRN_EMULATOR_INIT 16, 198
INCLUDE_SELECT 13
INCLUDE_UHCI 11, 178
INCLUDE_UHCI_INIT 12, 179
INCLUDE_USB 11, 174
INCLUDE_USB_AUDIO_DEMO 191
INCLUDE_USB_INIT 11, 175
INCLUDE_USB_KEYBOARD 12, 180
INCLUDE_USB_KEYBOARD_INIT 12, 180
INCLUDE_USB_MOUSE 12, 181
INCLUDE_USB_MOUSE_INIT 12, 182
INCLUDE_USB_MS_BULKONLY 12, 185
INCLUDE_USB_MS_BULKONLY_INIT 12, 186
INCLUDE_USB_MS_CBI 12, 187
INCLUDE_USB_MS_CBI_INIT 12, 188
INCLUDE_USB_PEGASUS_END 12, 189
INCLUDE_USB_PEGASUS_END_INIT 12, 190
INCLUDE_USB_PRINTER 12, 183
include_usb_printer_init 12, 183
include_usb_speaker 12, 184
include_usb_speaker_init 12, 27, 185
include_usb targ 15, 192
include_usbtool 13, 200
including the host stack component 11
index 124
featureClear() 119
featureSet() 119
statusGet() 124
vendorSpecific() 126
initialization
host controller 11, 23
host stack 23
USB devices 12
USB host stack 11
initializing USB devices 12
installDir 2
interface
interfaceGet() 123
interfaceSet() 123
interfaceGet() 123
described 123
table of associated control requests 113
interfaceSet() 123
described 123
table of associated control requests 113
interrupt behavior 102
interrupt endpoint 102
defined 208
interrupt levels 17
interrupt request 208
interrupt request line 208
interrupt request, see IRQ
interrupt routines
PCI and non-PCI 172
interrupt status 151
interrupt transfer 208
interrupt transfers 2
interrupt vectors 17
interval
USB_ENDPOINT_DESCR 129
ioBase 107
IP_MAX_UNITS 22
IRP 207
IRQ 107, 208
Irq data member
explained 161
irq element 107
isBandwidthAvailable() 61
isochronous transfer 208
isochronous transfers 2
isp1582 16
ISR
PCI and non-PCI routines 172
isRequestPending() 62
K
kbdDown 219
kbdInit 219
kbdPoll 219
command to pool the keyboard 220
example 221
kbdReport 220
keyboard 179
application testing 219
drivers 72
emulator 16, 194
initialization 25
queue size 181
keyboard drivers
data flow 78
initialization 73
ioctl routines 78
typematic repeat 79
Keyboard Init 180
L
languageId
descriptorGet() 120, 122
descriptorSet() 120, 122
length 124
descriptorGet() 122
descriptorSet() 122
statusGet() 124
USB_ENDPOINT_DESCR 128
vendorSpecific( ) 126
LINK 132
little endian 208
little-endian platforms
host stack tested on 157
locating the host controller 162
low data transfer rate 2
low-speed 208

M
management event codes
mngmtFunc( ), parameter to 113
TARG_MNGMT_ATTACHMENT 114
TARG_MNGMT_BUS_RESET 114
TARG_MNGMT_DISCONNECT 113
TARG_MNGMT_RESUME 113
TARG_MNGMT_SUSPEND 114
Mass Storage - Bulk 185
mass storage application
testing 223
Mass Storage - CBI 187
mass storage class device
initialization 25
mass storage class drivers 93
data flow 98
dynamic attachment 97
initialization 98
XBD driver model 94
mass storage devices
configuring bulk mass storage device names
setting device names 13
setting device names 14, 18, 19
Mass Storage Emulator
including components 16
reference 196
mass storage function driver 21
Mass Storage USB Driver- Control/Bulk/Interrupt component 187
maximum packet size 129
Maximum UFI Devices 188
maxPacketSize
USB_ENDPOINT_DESCR 129
message pipe 208
microframe 209
MIDI 88
MIDIStreaming 91
mngmtCode parameter
mngmtFunc( ), to 113
mngmtFunc( ) callback function 113
modifyDefaultPipe( ) 61
mouse 181
driver 80
initialization 25
mouse driver
data flow 83
dynamic attachment 82
initialization 80
ioctl routines 83
Mouse Init 182
mouseTest 219
MPEG 88
ms7727se
sample code 168
M-Systems FlashOnKey 26

N
NetChip NET2280 15, 193
USB Peripheral Controller Driver 192
networking components 22
node IDs 52
non-PCI buses
porting 166
non-PCI OHCI host controller
sample code for initialization 166
non-PCI resources
assignment of 166

O
OHCI 2, 33, 177
host controller initialization 11
Init 177
sample code for initialization 164
specification 7
OHCI_PGMIF host controller value 162
Open Host Controller Interface component 177
 Initialization component 177
Open Host Controller Interface, see OHCI OSAL
definition 36
OUT endpoint defined 209

P

pActLen
descriptorGet() 122
descriptorSet() 122
pAlternateSetting
interfaceGet() 123
param
callback routine parameter 112
parity errors 17
pBaseAddress data member explained 161
pBfr 124
descriptorGet() 122
statusGet() 124
USB_BFR_LIST 134
usbTargControlPayloadRecv() 136
pBuf
usbTargPipeStatusGet() 136
pBuffer
copying data to and from 151
TRB_COPY_DATA_TO/FROM_EPBUF 151
PCI buses
porting 164
PCI interface 36
PCI resources
porting and 164
pciConstants.h 159
pciFindClass() distinguishing from usbPciClassFind 162
pciFindClass() routine relationship to usbPciClassFind() routine 162
pciFindDevice() code example 106
pConfiguration 119
pContext
mgmtFunc() for TARG_MNGMT_ATTACH 114
mgmtFunc() for TARG_MNGMT_BUS_RESET 115
mgmtFunc() for TARG_MNGMT_DETACH 115
mgmtFunc() for TARG_MNGMT_DISCONNECT 116
mgmtFunc() for TARG_MNGMT_SUSPEND and TARG_MNGMT_RESUME 116
mgmtFunc() parameter 114
pContext parameter
mgmtFunc() parameter described 114
pDeviceInfo
TRB_ATTACH 146
pDeviceSpecificData data member explained 162
PDIUSB12 16
PDIUSB12D12 194
Pegasus Device Maximum Number 190
Pegasus IP Address 190
Pegasus Net Mask 190
Pegasus network device initialization 27
setting device parameters 14
setting device parameters in config.h 19
Pegasus Target Name 190
PEGASUS_IP_ADDRESS 190
PEGASUS_MAX_DEVS 190
PEGASUS_NET_MASK 190
PEGASUS_TARGET_NAME 190
pEndpointDesc
TRB_ENDPOINT_ASSIGN 149
peripheral stack, see USB peripheral stack pErp
usbTargTransferAbort() 135
pFrameNo
synchFrameGet() 125
usbTargCurrentFrameGet() 137
pFuncBufferSwap data member explained 161
pFuncBufferSwap() function pointer member
usbBufferSwap( ) implementation 171
pFuncBusToCpu data member explained 160
pFuncCpuToBus data member explained 160
pFuncDataSwap data member explained 160
pFuncIntConnect data member explained 160
pFuncIntDisconnect data member explained 161
pgmIf programming interface 162
pHalDeviceInfo 145
   TRB_ATTACH 146
pHalPipeHandle 127
Philips ISP1582 USB Peripheral Controller Driver 193
PhilipsIsp1582 193
pid
   USB_BFR_LIST 134
pipe 209
pipe handle 209
pipeHandle 127, 130, 149, 151
   TRB_COPY_DATA_TO/FROM_EPBUF 151
   TRB_ENDPOINT_ASSIGN 149
   TRB_ENDPOINT_RELEASE 149
   TRB_IS_BUFFER_EMPTY 151
usbTargPipeDestroy( ) 130
play 219
polling 209
polling interval 129
pollInput() 76
porting
   for PCI buses 164
   non-PCI buses 166
   PCI resources 164
power management 2
pPipeHandle
   USB_ERP 133
   usbTargPipeCreate( ) 127
print 219
print command
   printer application 223
print4k 219
eexample 222, 223
printer 182
   application testing 221
   emulator 16, 197
   initialization 25
printer driver 84
   data flow 87
   dynamic attachment 86
   IEEE-1284 87
   initialization 84
   ioctl routines 87
Printer Init 183
prnDown 219
prnDump 223
   command from target peripheral 222
   example 222
prnInit 219
programming interface pgmIf 162
project facility 10
protocol 209
pTargChannel
   usbTargTcdAttach( ) 112
pTargTcd 127
pTrb
   single entry point 142
pUSB_HCD_BUS_INFO data structure
   defined 159
   members 160
pUSB_IRP 102
registering client modules 38, 47
   USBHST_DEVICE_DRIVER data structure 38
registering the host controller driver 59
registering the hub class driver 68
release 210
Remote Wakeup
   setting and clearing a device 137
   supporting 145
   uFeatureSelector element 148
remote wakeup 210
removing a device from a hub 71
request
   vendorSpecific( ) 126
request types
  table of standard 117
requestType
descriptorGet() 120
descriptorSet() 120
featureClear() 117
featureSet() 117
statusGet() 124, 125
vendorSpecific() 126
reset 210
reset event 154
result
  USB_ERP 132, 133, 135
  resume 210
  resume event 154
  resume signal 113
  RESUME signaling 137
  root hub 210
  root hub emulation 65
  root port 210

S
S_usbTcdLib_BAD_HANDLE 132
S_usbTcdLib_BAD_PARAM 132
S_usbTcdLib_CANNOT_CANCEL 133
S_usbTcdLib_COMM_FAULT 133
S_usbTcdLib_DATA_OVERRUN 133
S_usbTcdLib_DATA_TOGGLE_FAULT 133
S_usbTcdLib_ERP_CANCELED 133
S_usbTcdLib_ERP_CANCELELED aborting a transfer 135
S_usbTcdLib_GENERAL_FAULT 133
S_usbTcdLib_HW_NOT_READY 133
S_usbTcdLib_INT_HOOK_FAILED 133
S_usbTcdLib_NEW_SETUP_PACKET 133
S_usbTcdLib_NOT_IMPLEMENTED 133
S_usbTcdLib_NOT_INITIALIZED 133
S_usbTcdLib_NOT_SUPPORTED 133
S_usbTcdLib_OUT_OF_MEMORY 132
S_usbTcdLib_OUT_OF_RESOURCES 133
S_usbTcdLib_PID_MISMATCH 133
S_usbTcdLib_SHUTDOWN 133
S_usbTcdLibSTALL_ERROR 133

sample code
  initializing PCI-based OHCI host controller
  initializing resources for non-PCI controllers
  initializing resources for non-PCI OHCI host controller
  initializing resources for PCI and non-PCI host controllers
  usbBufferSwap 172
  usbPciClassFind() 163, 168
  usbPciConfigHeaderGet() 106
SCSI Primary Commands
  2 (SPC-2), Revision 3 93
SCSI-10 commands 26
SCSI-6 commands 26
SEQ_DEV 88
SEQ_DEV driver model 88
SET_ADDRESS 125, 147
table of control requests 113
SET_CONFIGURATION 41
  request, response to 119
table of control requests 112
SET_DESCRIPTOR 41, 120, 122
table of control requests 112
SET_FEATURE 42, 137, 147
  featureSet() callback 116
table of control requests 112
SET_INTERFACE 41, 113, 123
setBitRate() 62
setting and clearing a device
  Remote Wakeup 137
setting configuration parameters 13
setup 210
setup packet
  defined 210
shared memory network parameters 17
single entry point 142
SIO driver model 72
SIO_CALLBACK_PUT_MOUSE_REPORT 83
SIO_CALLBACK_PUT_RCV_CHAR 83
SIO_CHAN 75
SIO_MODE_INT 87
SOF 211
speaker 184
Index

Init component 185
initialization 25
speaker driver
data flow 92
dynamic attachment 90
initialization 88
ioctl routines 91
overview 88
SEQ_DEV driver model 88
specifications
EHCI 7
host controller 2
OHCI 2, 7
UHCI 7
spkrDown 219
spkrFmt 219
spkrInit 219
stall 135, 210
STALL state 135
standard request 211
standard request interfaces 41
start-of-frame, see SOF
state 211
statusGet() 125
described 124
table of associated control requests 113
stub files
usbPciLib 171
submitURB() 61
suspend 211
suspend event 154
suspend signal 113
SUSPEND state 137, 147
SYNCH_FRAME 42, 125, 126
synchFrameGet 113
synchFrameGet() 125
table of associated control requests 113
synchronization type 128
system clock 17

T
Targ Keyboard Emulator Driver Initialization 195
Targ Keyboard Emulator Init 195
Targ Mass Storage Emulator Init 196
Targ Printer Emulator Init 198
TARG_MNGMT_ATTACH
management code parameters 113
usage described
TARG_MNGMT_DETACH
management code parameters 113
usage described
TARG_MNGMT_BUS_RESET
management code parameters 113
TARG_MNGMT_BUS_RESET management event code 115
TARG_MNGMT_CONNECT
management code parameters 113
usage described
TARG_MNGMT_DETACH
management code parameters 113
usage described
TARG_MNGMT_RESUME 116
management code parameters 113
usage described
TARG_MNGMT_SUB_RESET
USB_TCD_FULL_SPEED, parameter to 115
USB_TCD_HIGH_SPEED, parameter to 115
USB_TCD_LOW_SPEED, parameter to 115
TARG_MNGMT_SUSPEND
management code parameters 113
usage described
TARG_PIPE 126
TARG_TCD 132
targCallback() 133
USB_ERP 133
targChannel, callback routine parameter 112
target application 5, 211
attaching to TCD 5
callback table 108
target channel 211
defined 211
target controller 211
target controller driver, see TCD
target controller, see TC
target layer 212
functions of 5
initializing 105
overview 5
uninitializing 105
usage count 105

235
target request block, see TRB

targInit kbd
  example 220
  usage 219

targInit prn
  example 221
  usage 221

targLink
  USB_ERP 132

targPtr
  USB_ERP 132

TC 211

TCD 5
  attaching to target application 5
  enabling and disabling 110
  initialization 5
  overview of responsibilities 5
  single entry point 142

Target Controller Driver, abbreviation for 5

TCD handle
  USB_TARG_CHANNEL, stored in 105

TCD_ENDPOINTSTALL 135, 150
TCD_ENDPOINTUNSTALL 135, 136, 150
TCD_FNC_ADDRESS_SET 143, 147
TCD_FNC_ATTACH 107, 143, 144, 145
TCD_FNC_COPY_DATA_FROM_EPBUF 135, 143, 150
TCD_FNC_COPY_DATA_TO_EPBUF 135, 143, 150
TCD_FNC_CURRENT_FRAME_GET 137, 143, 148
TCD_FNC_DETACH 107, 143, 146
TCD_FNC_DEVICE_FEATURE_CLEAR 143, 147
  calling the TCD entry point with 138
TCD_FNC_DEVICE_FEATURE_SET 143, 147
  calling entry point of TCD with 137
TCD_FNC_DISABLE 111, 143, 146
TCD_FNC_ENABLE 110, 143, 146
TCD_FNC_ENDPOINT_ASSIGN 127, 143, 149
TCD_FNC_ENDPOINT_INTERRUPT_STATUS_CLEAR 144, 154
TCD_FNC_ENDPOINT_INTERRUPT_STATUS_GET 144, 152
TCD_FNC_ENDPOINT_RELEASE 130, 143, 149
TCD_FNC_ENDPOINT_STATE_SET 135, 143, 149
TCD_FNC_ENDPOINT_STATUS_GET 136, 143, 150
TCD_FNC_HANDLE_DISCONNECT_INTERRUPT 144, 154
TCD_FNC_HANDLE_RESET_INTERRUPT 144, 154
TCD_FNC_HANDLE_RESUME_INTERRUPT 144, 154
TCD_FNC_HANDLE_SUSPEND_INTERRUPT 144, 154
TCD_FNC_INTERRUPT_STATUS_CLEAR 144, 151, 152
TCD_FNC_INTERRUPT_STATUS_GET 144, 151, 152
TCD_FNC_IS_BUFFER_EMPTY 143, 151
TCD_FNC_RESUME 147
TCD_FNC_SIGNAL_RESUME 137, 143
TCD-defined parameters 106

tcdLink
  USB_ERP 132
tcdParam
  TRB_ATTACH 145
tcdPtr
  USB_ERP 132

Test Mode
  feature cannot be cleared 138
  setting and clearing a device feature 137
  uFeatureSelector element specifies the 148
  whether a device supports 145

test mode 212
test selectors 138, 148
token packet 212
toolchain 212
transaction 212
transfer type 128, 212
transferType
  USB_ERP 134
translation layer 35
translation unit 212

TRB 212
  description of 142
  specific codes in a 142

TRB header 142

trbLength
  TRB header 144
tUsbdCln 48
tUsbKbd 79
tUsbMSCxbdStrategy task 96
typematic repeat 79

U

uActLength
  TRB_COPY_DATA_TO/FROM_EPBUF 151
uDeviceFeature
  USB_APPLN_DEVICE_INFO 145
  USB_APPLN_DEVICE_INFO structure member 114
uEndpointInterruptStatus
  TRB_ENDPOINT_INTERRUPT_STATUS_GET 153
uEndpointNumberBitmap
  USB_APPLN_DEVICE_INFO 114, 145
uFeatureSelector
  TRB_DEVICE_FEATURE_SET_CLEAR 148
  uFeatureSelector
    usbTargDeviceFeatureSet() 137
UFI Device Name Size 189
UFI_MAX_DEVS 188
UFI_MAX_DRV_NAME_SZ 189
UHCI 2, 33, 178, 213
  host controller initialization 11
  specification 7
UHCI_Init 179
UHCI_PGMIF host controller value 162
uInterruptStatus 151
  TRB_INTERRUPT_STATUS_GET_CLEAR 151
Universal Host Controller Interface component 178
Universal Host Controller Interface Initialization component 179
Universal Host Controller Interface, see UHCI
Universal Serial Bus Driver, see USBD
Universal Serial Bus Specification 7
un-stall 135
uNumberEndpoints
  USHHAL_DEVICE_INFO 145
URB 213

usage type 129
USB 1.0
  compatibility 3
USB 2.0
  compatibility 3
  endian conversion 166
  technology overview 2
USB 2.0 host controller driver 36
USB 2.0 hub class driver 35
USB 2.0 translation layer 35
USB 2.0 USBD layer 35
USB Audio Demo 191
USB Audio Demo for Host Stack component 191
USB Bulk Device Name Size 187
USB Bulk Drive Name 186
USB Bulk Maximum Drives 187
USB Bulk Non-Removable Disk 187
USB CBI Drive Name 188
USB class drivers 34
  configuring in config.h 18, 19
USB classes 49
USB client module 34
USB D12 Emulator 199
USB data toggles 53
USB device 213
USB devices
  components 10
  initialization 12
  locating components in the kernel editor 10
USB Floppy Interface (UFI) Command Specification 93
USB host controller drivers 33
overview 33
USB host driver 33, 35
  see also USB host stack configuring in config.h 17
  initializing 37, 45
  USB 1.0 compatibility 45
  USBD 3
USB host modules 34
USB host stack
  architecture 31
  component 174
component dependencies 21
component reference summary 174
configuration 8
configuring
  with the project facility 10
configuring and building VxWorks 9
configuring using the command line 16
including from Workbench 11
Init component 175
initialization 11
Initialization component 175
overview 3
setting configuration parameters 13
USB Keyboard Class Driver
  Initialization component 180
USB Keyboard Driver
  component 179
USB Mass Storage- CBI Class Driver Initialization component 188
USB Mass Storage Emulator Firmware 194
USB Mouse Driver
  component 181
  Initialization component 182
USB Pegasus END driver 22
USB Peripheral Controllers 15
Usb Peripheral Init 16
USB peripheral stack 15, 192, 209
  including the component 15
  reference 192
USB Printer Driver
  component 182
  Initialization component 183
USB protocols 49
USB Request Block, see URB
USB Speaker Driver
  component 184
  Initialization component 184
USB speakers
  compound devices 88
  recognizing and handling 91
USB subclasses 49
USB Tool
  component 199
USB Tool - Test Application for Host Stack 199
USB UFI Device Name Size 189
usb.h 49
USB_APPLN_DEVICE_INFO
  structure elements 114
  TCD populating 145
USB_APPLN_DEVICE_INFO structure
  mngmtFunc( ), parameter to 114
USB_BFR_LIST 134
USB_DATA0 134
USB_DATA1 134
USB_DESCR_CONFIGURATION 121
USB_DESCR_DEVICE 121
USB_DESCR_DEVICE_QUALIFIER 122
USB_DESCR_ENDPOINT 122
descriptorType 128
USB_DESCR_INTERFACE 122
USB_DESCR_INTERFACE_POWER 122
USB_DESCR_OTHER_SPEED_CONFIGURATION 122
USB_DESCR_STRING 121
USB_DEV_STS_LOCAL_POWER 124
USB_DEV_STS_REMOTE_WAKEUP 124
USB_ENDPOINT_DESCR 128
USB_ENDPOINT_STS_HALT 125, 136
USB_ERP 132
  size 132
USB_ERP structure 130
USB_FEATURE_DEVICE_REMOTE_WAKEUP 114, 145
USB_FEATURE_TEST_MODE 114, 145
USB_FEATURE_USB20 114, 145
USB_FSEL_DEV_ENDPOINT_HALT 118
USB_FSEL_DEV_REMOTE_WAKEUP 118
USB_FSEL_DEV_TEST_MODE 118
USB_IRP 58
USB_KBD_ATTACH 75
USB_KBD_REMOVE 75
USB_MSC_ATTACH 97
USB_MSC_DETACH 97
USB_MSC_DETACH 82
USB_MSC_DETACH 82
USB_NODE_ID 52
USB_PID_IN 134
USB_PID_OUT 134
USB_PID_SETUP 134
USB_PRN_ATTACH 86
USB_PRN_REMOVE 86
USB_REQ.Clear_FEATURE 117
USB_REQ.Get_CONFIGURATION 117
USB_REQ.Get_DESCRIPTOR 117
USB_REQ.Get_INTERFACE 117
USB_REQ.Get_STATE 117
USB_REQ.Get_STATUS 117
USB_REQ.Set_ADDRESS 117
USB_REQ.Set_CONFIGURATION 117
USB_REQ.Set_DESCRIPTOR 117
USB_REQ.Set_FEATURE 117
USB_REQ.Set_INTERFACE 117
USB_REQ.Set_FEATURE_Request described 118
USB_REQ.Set_INTERFACE 117
USB_REQ.Synch_FRAME 117
USB_RT_CLASS 121
USB_RT.DEV_TO_HOST 121
USB_RT.DEVICE 121
USB_RT.ENDPOINT 121
USB_RT.HOST_TO_DEV 121
USB_RT.INTERFACE 121
USB_RT.OTHER 121
USB_RT.STANDARD 121
USB_RT.VENDOR 121
USB_SCSI_FLAG.READ_WRITE10 26
USB_SCSI_FLAG.READ_WRITE6 26
USB_SPKR_ATTACH 90
USB_SPKR_REMOVE 90
USB_TARG.Address_SET_FUNC 109
USB_TARG_CHANNEL parameter to callback routines storing the TCD handle 105
USB_TARG_CONFIGURATION.GET_FUNC 108
USB_TARG_CONFIGURATION.SET_FUNC 109
USB_TARG.DESCRIPTOR.GET_FUNC 109
USB_TARG.DESCRIPTOR.SET_FUNC 109
USB_TARG_FEATURE.CLEAR_FUNC 108
USB_TARG_FEATURE.SET_FUNC 108
USB_TARG_INTERFACE.GET_FUNC 109
USB_TARG_INTERFACE.SET_FUNC 109
USB_TARG.MANAGEMENT_FUNC 108
USB_TARG_STATUS.GET_FUNC 109
USB_TARG.Sync_FRAME.GET_FUNC 109
USB_TARG_VENDOR_SPECIFIC_FUNC 109
USB_TCD.FULL_SPEED TARG_MNGMT_BUS_RESET 115
USB_TCD.HIGH_SPEED TARG_MNGMT_BUS_RESET 115
USB_TCD.LOW_SPEED TARG_MNGMT_BUS_RESET 115
USB_TEST_MODE.J_STATE 138
USB_TEST_MODE.K_STATE 138
USB_TEST_MODE.SEO_ACK 138
USB_TEST_MODE.TEST_FORCE_ENABLE 138
USB_TEST_MODE.TEST_PACKET 138
USB_XFRTYPE.BULK 134
USB_XFRTYPE.CONTROL 134
USB_XFRTYPE INTERRUPT 134
USB_XFRTYPE_ISOCH 134
USB.FR создания 101
USB.FR_DELETE 101
USB.FR.DEV 101
usbAcmLib 35
usbAudio demo initialization 27
usbBufferSwap sample implementation 172
usbBufferSwap() explained 161
usbBulkDevLib 35, 186
usbChUfiDevLib 35, 187
USBD 213
USBD 1.1 45
USBD 2.0 36
USBD, see USB host driver
USBD_PIPE.Handle 55
usbDataSwap() 160
usbDevCountGet() 52
usbClientRegister() 48
usbConfigurationSet() 53
usbDynamicAttachRegister() 49
usbHubInit() 37
usbHubPortCountGet() 52
usbInit() 37
usbInitialize() 45
usbLib 52
usbLib.h 49
usbNodeIdGet() 52
usbNodeInfoGet() 52
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>usbdPciInit()</td>
<td>174</td>
</tr>
<tr>
<td>usbdRootNodeIdGet()</td>
<td>52</td>
</tr>
<tr>
<td>usbdTargTransfer()</td>
<td>130</td>
</tr>
<tr>
<td>usbEnum</td>
<td></td>
</tr>
<tr>
<td>command at usbTool prompt</td>
<td>220</td>
</tr>
<tr>
<td>example</td>
<td>220</td>
</tr>
<tr>
<td>USBHAL_DEVICE_INFO</td>
<td>145</td>
</tr>
<tr>
<td>USBHAL_PIPE_INFO</td>
<td>148</td>
</tr>
<tr>
<td>usbHalIsr()</td>
<td>145</td>
</tr>
<tr>
<td>usbHalIsrParam</td>
<td>145</td>
</tr>
<tr>
<td>TRB_ATTACH</td>
<td>145</td>
</tr>
<tr>
<td>USBHST_BAD_START_OF_FRAME</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_BUFFER_OVERFLOW_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_BUFFER Underrun_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_DATA_OVERFLOW_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_DATA Underrun_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_HC_DRIVER</td>
<td>59</td>
</tr>
<tr>
<td>USBHST_HC DRIVER data structure defined</td>
<td>38</td>
</tr>
<tr>
<td>USBHST_DEVICE NOT RESPONDING_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_HC DRIVER</td>
<td>59</td>
</tr>
<tr>
<td>USBHST_INSUFFICIENT_BANDWIDTH</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_INSUFFICIENT_RESOURCE</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_INVALID_PARAMETER</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_MEMORY_NOT_ALLOCATED</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_STALL_ERROR</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_TIMEOUT</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_TRANSFER_CANCELLED</td>
<td>65</td>
</tr>
<tr>
<td>USBHST_UCB</td>
<td>43</td>
</tr>
<tr>
<td>usbHstBusDeregister()</td>
<td>64</td>
</tr>
<tr>
<td>usbHstBusRegister()</td>
<td>63</td>
</tr>
<tr>
<td>usbHstClearFeature()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstDriverDeregister()</td>
<td>40, 71</td>
</tr>
<tr>
<td>usbHstDriverRegister()</td>
<td>68</td>
</tr>
<tr>
<td>usbHstGetConfiguration()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstGetDescriptor()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstGetTime()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstHCDRegiister()</td>
<td>64</td>
</tr>
<tr>
<td>usbHstHCDRegister()</td>
<td>63</td>
</tr>
<tr>
<td>usbHstSetConfiguration()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstSetDescriptor()</td>
<td>41</td>
</tr>
<tr>
<td>description example</td>
<td>42</td>
</tr>
<tr>
<td>example</td>
<td>42</td>
</tr>
<tr>
<td>usbHstSetFeature()</td>
<td>42</td>
</tr>
<tr>
<td>usbHstInterface()</td>
<td>41</td>
</tr>
<tr>
<td>usbHstSetSynchFrame()</td>
<td>42</td>
</tr>
<tr>
<td>usbHstURBCancel()</td>
<td>43</td>
</tr>
<tr>
<td>usbHstURBSubmit()</td>
<td>43</td>
</tr>
<tr>
<td>data transfer interface example</td>
<td>43</td>
</tr>
<tr>
<td>usbi</td>
<td>222</td>
</tr>
<tr>
<td>usbInit</td>
<td></td>
</tr>
<tr>
<td>command for usbTool</td>
<td>218</td>
</tr>
<tr>
<td>example</td>
<td>175</td>
</tr>
<tr>
<td>usbdPciClassFind()</td>
<td></td>
</tr>
<tr>
<td>defined in usbPciStub.c</td>
<td>162</td>
</tr>
<tr>
<td>distinguishing from pciFindClass()</td>
<td>162</td>
</tr>
<tr>
<td>sample implementation for ms7727se</td>
<td>168</td>
</tr>
<tr>
<td>sample implementation</td>
<td>163</td>
</tr>
<tr>
<td>usbPciConfigHeaderGet()</td>
<td>106</td>
</tr>
<tr>
<td>sample code</td>
<td>106</td>
</tr>
<tr>
<td>usbPciIntConnect()</td>
<td></td>
</tr>
<tr>
<td>explained</td>
<td>160</td>
</tr>
<tr>
<td>usbPciIntRestore()</td>
<td></td>
</tr>
<tr>
<td>explained</td>
<td>161</td>
</tr>
</tbody>
</table>
Index

usbPciLib 36
    modifying for PCI and non-PCI routines 158
usbPciStub.c
    modifying for PCI and non-PCI routines 158
    usbPciClassFind() defined in 162
usbPegasusEndLib 35
usbPrinterDevInit() 84
usbPrinterDynamicAttachRegister() 86
usbPrinterLib 35, 84, 183
usbPrinterSioChanLock() 87
usbPrinterSioChanUnlock() 87
usbSpeakerDevInit() 88
usbSpeakerDynamicAttachRegister() 90
usbSpeakerLib 35, 88, 184
usbSpeakerLib sd_seqWrt() 92
usbSpeakerSeqDevLock() 90
usbSpeakerSeqDevUnlock() 90
usbTargControlPayloadRecv() 126, 128, 130, 136
usbTargControlResponseSend() 126, 128, 130, 136
usbTargControlStatusSend() 126, 128, 130, 136
usbTargCurrentFrameGet() 137
usbTargDeviceFeatureClear() 137
usbTargDeviceFeatureSet() 137
usbTargDisable()
    enabling and disabling the TCD 111
usbTargEnable() 110
    enabling and disabling the TCD 110
usbTargEnable() routine
    attaching a TCD 106
usbTargInitialize() 105
usbTargKbdLib 195
usbTargLib.h 113
    callback table defined in 112
usbTargMsLib 196
usbTargPhilipsD12EvalLib 199
usbTargPipeCreate() 127, 130
usbTargPipeDestroy() 127
usbTargPipeStatusGet() 136
usbTargPipeStatusSet() 135
usbTargPrnLib 198
usbTargRbcLib 196
usbTargShutdown() 105
usbTargSignalResume() 137
usbTargStatusGet() 136
usbTargTcdAttach() 107, 110, 145
enabling and disabling the TCD 110
establishing communication 107
prototype 105
usbTargTcdDetach() 107
prototype 107
usbTargTransfer() 130
    aborting a transfer submitted by 135
transferring data 130
usbTargTransferAbort() 135
    prototype 135
usbTcd.h 153
USBTCRD_ENDPOINT_BIT_STUFF_ERROR 153
USBTCRD_ENDPOINT_COMMUN_FAULT 153
USBTCRD_ENDPOINT_CRC_ERROR 153
USBTCRD_ENDPOINT_DATA_OVERRUN 153
USBTCRD_ENDPOINT_DATA_TOGGLE_ERROR 153
USBTCRD_ENDPOINT_DATA_UNDERRUN 153
USBTCRD_ENDPOINT_PID_MISMATCH 153
USBTCRD_ENDPOINT_STALL_ERROR 153
USBTCRD_ENDPOINT_TIMEOUT_ERROR 153
USBTCRD_ENDPOINT_TRANSFER_SUCCESS 153
usbTool 217
    attach ehci 218
    attach ohci 218
    attach uhci 218
cautions with initialization components 25
code exerciser utility 217
commands 218
configuring in config.h 21
exection sequence 218
keyboard test commands 219
mouse test commands 219
printer test commands 219
reference summary 199
relationship to other modules 34
running from the shell 218
speaker test commands 219
testing devices 218
testing the keyboard application with 219
testing the printer application with 221
usblInit 218
usbTool Test Application
    using 217
usbXXXDevLock() 101

241
usbXXXDevUnlock() 101
usbXXXDynamicAttachRegister() 101
usbXXXEndInit() 101
usbXXXEndLib 99
userCallback() 133
    aborting a transfer 135
    and usbTargControlPayloadRecv() pBuffer parameter 136
    required for result field 133
USB_BRF_LIST structure, actLen 134
USB_ERP 133
userPtr, USB_ERP 132
usrUsbAudioDemo() 191
usrUsbAudioDemo.c 191
usrUsbBulkDevInit() 186
usrUsbBulkDevInit.c 186
usrUsbCbiUfiDevInit() 188
usrUsbCbiUfiDevInit.c 188
usrUsbHcdEhciAttach() 176
usrUsbHcdEhciAttach.c 176
usrUsbHcdOhciAttach() 178
usrUsbHcdOhciAttach.c 178
usrUsbHcdUhciAttach() 179
usrUsbHcdUhciAttach.c 179
usrUsbInit.c 175
usrUsbKbdInit() 180
usrUsbKbdInit.c 180
usrUsbMseInit() 182
usrUsbMseInit.c 182
usrUsbPciInit.c 174
usrUsbPegasusEndInit() 190
usrUsbPegasusEndInit.c 190
    initializing Pegasus class driver 27
usrUsbPrmInit() 183
usrUsbPrmInit.c 183
usrUsbSpkrInit() 185
usrUsbSpkrInit.c 185
usrUsbTool.c 200
uTestSelector 148
    TRB_DEVICE_FEATURE_SET_CLEAR 148
    usbTargDeviceFeatureSet() 137
utilities, usbTool 217

V
value, vendorSpecific() 126
vendorSpecific 113
vendorSpecific() 126
table of associated control requests 113
volume 219
VxWorks
    command line configuration 16
    component dependencies 21
    config.h 16
    configuring components 9
    header files 16
    project facility 10
    USB device components 10

W
wakeup 213
Wind River Workbench 10
wIndex, vendorSpecific() 126
wLength 124
    statusGet() 124
    vendorSpecific() 126
wValue 126

X
XBD
device driver model 94
driver model 94
XXXEndRecv() 102
XXXEndSend() 102

Z
zero-length packet 132