

Wind River[®] USB for VxWorks[®] 6

PROGRAMMER'S GUIDE

2.4

Copyright © 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

<http://www.windriver.com/company/terms/trademark.html>

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:

installDir\product_name\3rd_party_licensor_notice.pdf.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

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

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

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

<http://www.windriver.com>

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

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

Contents

1	Overview	1
1.1	Introduction	1
1.2	Technology Overview	2
1.3	USB Component Overview	2
1.3.1	USB Host Stack	3
	Host Controller Drivers	3
	USBD and Class Drivers	4
1.3.2	USB Peripheral Stack	4
1.3.3	Architecture and BSP Support	6
1.3.4	SMP Ready	6
1.4	Additional Documentation	7
1.4.1	USB Specification Information	7
1.4.2	Peripheral Stack Information	8
1.4.3	Configuration Information	8
1.4.4	Latest Release Information	9
2	Configuring and Building Wind River USB	11
2.1	Introduction	11

2.2	Configuring and Building Wind River USB	11
2.3	Configuring VxWorks with Wind River USB	12
2.3.1	USB Host Stack Components and Parameters	12
	Parameters and Default Values	14
	Required Components	16
	Optional Components	16
	Component Dependencies	18
	Managing Dependencies	18
2.3.2	USB Peripheral Stack Components and Parameters	20
	Required Components	21
	Optional Components	21
	USBTool Components and Parameters	21
2.4	Building VxWorks with Wind River USB	22
2.5	Initializing USB Hardware	22
2.5.1	Initializing the USB Host Stack Hardware	22
	Startup Routines	23
	USBD Initialization	23
	Attaching the EHCI, OHCI, and UHCI Host Controllers	23
	Initialization Dependencies	24
	Keyboard, Mouse, Printer, and Speaker Initialization	25
	Mass Storage Class Device Initialization	25
	SCSI-6 Commands	26
	Communication Class Device Initialization	26
	USB Audio Demo Initialization	26
2.5.2	Initializing the USB Peripheral Stack Hardware	27
3	USB Host Drivers	29
3.1	Introduction	29
3.2	Architecture Overview	29
3.2.1	Host Controller Drivers and USBD	31
3.2.2	Class Drivers	32
3.2.3	Host Module Roadmap	32

3.3	The USB Host Driver	34
3.3.1	USB 2.0	34
	Initializing the USB	34
	Order of Initialization	35
	Bus Tasks	36
	Registering Client Modules	36
	Standard Request Interfaces	39
	Data Transfer Interfaces	40
3.3.2	USB 1.1 Compatibility	42
	Registering Client Modules	43
	Client Callback Tasks	43
	Dynamic Attachment Registration	44
	Device Configuration	48
	Pipe Creation and Deletion	50
	Data Flow	51
3.4	Host Controller Drivers	54
3.4.1	Registering the Host Controller Driver	54
3.4.2	USBHST_HC_DRIVER Structure	54
3.4.3	Host Controller Driver Interfaces	56
	USBHST_HC_DRIVER Structure	56
	USBHST_USBD_TO_HCD_FUNCTION_LIST Structure	57
3.4.4	Registering a Bus for the Host Controller	59
3.4.5	Deregistering the Bus for the Host Controller	59
3.4.6	Deregistering the Host Controller Driver	59
3.4.7	HCD Error Reporting Conventions	59
3.4.8	Root Hub Emulation	60
4	USB Class Drivers	61
4.1	Introduction	61
4.2	Hub Class Driver	62
4.2.1	Registering the Hub Class Driver	62
4.2.2	Connecting a Device to a Hub	64

4.2.3	Removing a Device From a Hub	64
4.2.4	Deregistering the Hub Class Driver	65
4.3	Keyboard Driver	65
4.3.1	SIO Driver Model	66
4.3.2	Initializing the Keyboard Class Driver	66
4.3.3	Registering the Keyboard Class Driver	67
4.3.4	Dynamic Device Attachment	68
4.3.5	ioctl Routines	71
4.3.6	Data Flow	71
4.3.7	Typematic Repeat	72
4.3.8	Uninitializing the Keyboard Class Driver	72
4.4	Mouse Driver	73
4.4.1	SIO Driver Model	73
4.4.2	Initializing the Mouse Class Driver	73
4.4.3	Registering the Mouse Class Driver	74
4.4.4	Dynamic Device Attachment	75
4.4.5	ioctl Routines	76
4.4.6	Data Flow	76
4.4.7	Uninitializing the Mouse Class Driver	76
4.5	Printer Driver	77
4.5.1	SIO Driver Model	77
4.5.2	Initializing the Printer Driver	77
4.5.3	Registering the Printer Driver	78
4.5.4	Dynamic Device Attachment	79
4.5.5	ioctl Routines	80
4.5.6	Data Flow	80
4.6	Audio Driver	80

4.6.1	SEQ_DEV Driver Model	81
4.6.2	Initializing the Audio Driver	81
4.6.3	Registering the Audio Driver	82
4.6.4	Dynamic Device Attachment	83
4.6.5	Recognizing and Handling USB Speakers	84
	Dynamic Attachment and Removal of Speakers	84
	Data Flow	85
4.6.6	Recognizing and Handling USB Microphones	85
	Data Flow	86
4.7	Mass Storage Class Driver	86
4.7.1	Extended Block Device Driver Model	88
4.7.2	API Routines	90
4.7.3	Dynamic Attachment	90
4.7.4	Initialization	91
4.7.5	Data Flow	91
4.8	Communication Class Drivers	92
4.8.1	Ethernet Networking Control Model Driver	92
4.8.2	Enhanced Network Driver Model	94
4.8.3	Dynamic Attachment	94
4.8.4	Initialization	95
4.8.5	Interrupt Behavior	95
4.8.6	ioctl Routines	96
4.8.7	Data Flow	96
5	USB Peripheral Stack Target Layer Overview	97
5.1	Introduction	97
5.2	Initializing the Target Layer	99
5.3	Attaching and Detaching a TCD	99

5.3.1	TCD-Defined Parameters	100
5.3.2	Detaching a TCD	101
5.3.3	Target Application Callback Table	102
5.4	Enabling and Disabling the TCD	104
5.5	Implementing Target Application Callback Routines	106
5.5.1	Callback and Target Channel Parameters	106
5.5.2	Control Pipe Request Callbacks	106
5.5.3	mngmtFunc() Callback	107
	Management Code Parameter	107
	Context Value Parameter	108
	Management Event Codes	108
5.5.4	Clear and Set Callbacks	110
	Request Type Parameter	111
	Feature Parameter	112
	Index Parameter	113
5.5.5	configurationGet() Callback	113
5.5.6	configurationSet() Callback	114
5.5.7	descriptorGet() and descriptorSet() Callbacks	114
	Request Type Parameter	115
	Descriptor Type and Index Parameters	116
	Language ID Parameter	116
	Length and Buffer Parameters	116
5.5.8	interfaceGet() Callback	117
5.5.9	interfaceSet() Callback	117
5.5.10	statusGet() Callback	118
5.5.11	addressSet() Callback	119
5.5.12	synchFrameGet() Callback	119
5.5.13	vendorSpecific() Callback	120
5.6	Pipe-Specific Requests	121
5.6.1	Creating and Destroying the Pipes	121

Endpoint Descriptor	122
usbTargPipeDestroy()	124
5.6.2 Transferring and Aborting Data	124
USB_ERP Structure	126
usbTargTransfer() Routine	129
Aborting a Data Transfer	129
5.6.3 Stalling and Unstalling the Endpoint	130
5.6.4 Handling Default Pipe Requests	130
5.7 Device Control and Status Information	131
5.7.1 Getting the Frame Number	131
5.7.2 Resuming the Signal	131
5.7.3 Setting and Clearing a Device Feature	132
5.8 Shutdown Procedure	133
6 Target Controller Drivers	135
6.1 Introduction	135
6.2 Hardware Adaptation Layer Overview	136
6.3 Single Entry Point	136
6.4 Target Request Block	136
6.5 Function Codes	138
6.5.1 Attaching the TCD	138
6.5.2 Detaching the TCD	140
6.5.3 Enabling and Disabling the TCD	140
6.5.4 Setting the Address	141
6.5.5 Resuming the Signal	141
6.5.6 Setting and Clearing the Device Feature	141
6.5.7 Getting the Current Frame Number	142
6.5.8 Assigning the Endpoints	142

6.5.9	Releasing the Endpoints	143
6.5.10	Setting the Endpoint Status	143
6.5.11	Getting the Endpoint Status	144
6.5.12	Submitting and Cancelling ERPs	144
6.5.13	Determining Whether the Buffer is Empty	145
6.5.14	Getting and Clearing Interrupts	145
6.5.15	Retrieving an Endpoint-Specific Interrupt	146
6.5.16	Clearing All Endpoint Interrupts	148
6.5.17	Handling Disconnect, Reset, Resume, and Suspend Interrupts	148
7	BSP Porting	151
7.1	Configuring the USB Peripheral Stack	151
7.1.1	Initialization of the USB Peripheral Stack	151
	Example Initializing Resources for a NET2280 controller	152
7.1.2	Creating a BSP-Specific Stub File for the USB Peripheral Stack	152
	Eight-, 16- and 32-Bit Data I/O	152
	Interrupt Routines	153
9	usbTool Code Exerciser Utility Tool	155
9.1	Introduction	155
9.2	Running usbTool from the Shell	156
9.3	Using the usbTool Execution Sequence	156
9.4	Testing Applications	157
9.4.1	Testing the Keyboard Application	157
9.4.2	Testing the Printer Application	159
9.4.3	Testing the Mass Storage Application	161

A	Glossary	163
A.1	Glossary Terms	163
A.2	Abbreviations and Acronyms	176

1

Overview

1.1	Introduction	1
1.2	Technology Overview	2
1.3	USB Component Overview	2
1.4	Additional Documentation	7

1.1 Introduction

This manual, the *Wind River USB Programmer's Guide*, covers 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, Workbench, and the 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.



NOTE: All file and directory paths in this manual are relative to the VxWorks installation directory. For installations based on VxWorks 5.x, this corresponds to *installDir*. For installations based on VxWorks 6.x, this corresponds to *installDir/vxworks-6.x*.

1.2 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
- bulk
- interrupt
- isochronous

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 different kinds of devices, each kind requiring its own device driver support. It is desirable that these device drivers be written to be 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 prebuilt USB class drivers, each of which handles one kind of USB device.

1.3 USB Component Overview

This section summarizes the Wind River USB host and peripheral stack components.

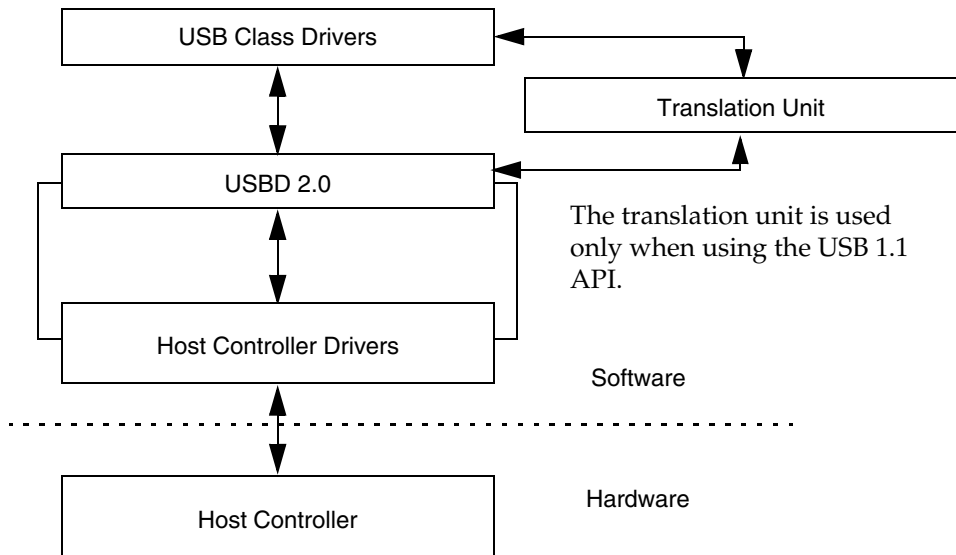
1.3.1 USB Host Stack

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 prebuilt USB host controller drivers (HCDs) for all three specifications as part of its USB host stack product.

Figure 1-1 Host Stack Overview



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 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:

- hub
- keyboard
- mouse
- printer
- audio
- 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.29.

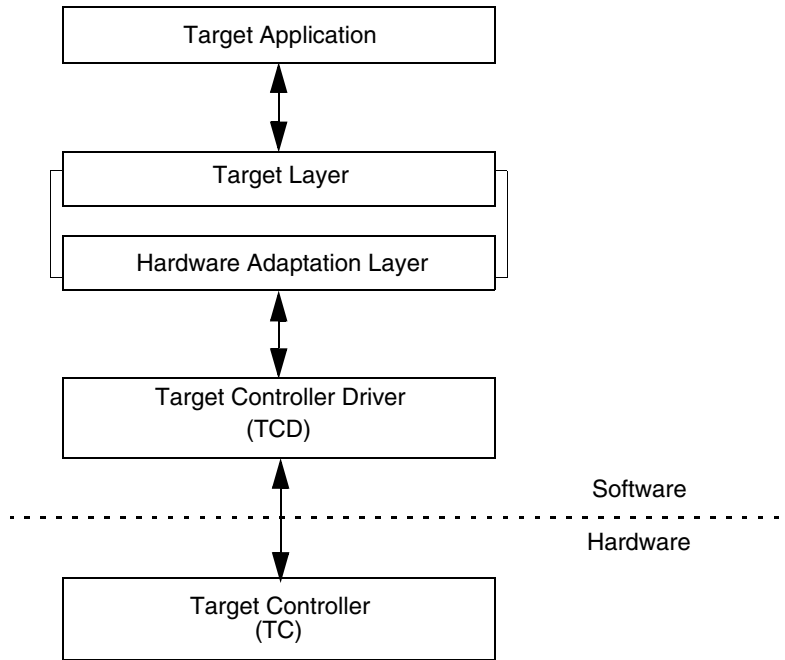
1.3.2 USB Peripheral Stack

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 USB peripheral stack.

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)

Figure 1-2 Peripheral Stack Overview



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. It makes it easier to port the USB peripheral stack 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 transfers with the default control pipe
- to provide interfaces to the target application for creating and deleting pipes to the endpoints
- to provide interfaces with the control and the non-control pipes for data transfers

At the top of the USB 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](#).

1.3.3 Architecture and BSP Support

The 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.

1.3.4 SMP Ready

VxWorks 6.6 introduces SMP facilities as a separately purchased product that support symmetric multiprocessing (SMP). Wind River USB for VxWorks 6 is SMP-ready, meaning that it runs correctly on SMP hardware, although it may not make use of more than one CPU.

For more technical information on SMP, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*. For purchasing details, please contact your local Wind River representative.

1.4 Additional Documentation

The following sections describe additional documentation about the technologies described in this book.

Reference pages for USB host stack libraries and routines 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.

1.4.1 USB Specification Information

For detailed specification information, refer to the following sources:

- *Universal Serial Bus Specification*, Rev. 2.0, April 27, 2000. All USB specifications are available at: <http://www.usb.org/developers/docs>
- USB device class specifications. All USB specifications are available at: http://www.usb.org/developers/devclass_docs#approved
 - *USB Mass Storage Class Specification Overview*, Rev. 1.2, June 23, 2003
http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf
 - *USB Mass Storage Class Bulk Only Transport*, Rev. 1.0, September 31, 1999
http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf
 - *USB Mass Storage Class Control/Bulk/Interrupt (CBI) Transport*, Rev. 1.1, June 23, 2003
http://www.usb.org/developers/devclass_docs/usb_msc_cbi_1.1.pdf
 - *USB Mass Storage Class UFI Command Specification*, Rev. 1.0, December 14, 1998
http://www.usb.org/developers/devclass_docs/usbmass-ufi10.pdf
 - *USB Device Class Definition for Printing Devices*, Rev. 1.1, January 2000
http://www.usb.org/developers/devclass_docs/usbprint11.pdf
 - *USB Device Class Definition for Human Interface Devices (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 (Search for UHCI from this page)
<http://www.intel.com>
- for UHCI errata on USB bandwidth reclamation, see page 24 in:
<ftp://download.intel.com/design/chipsets/specupdt/29773817.pdf>

1.4.2 Peripheral Stack Information

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

- *PDIUSB12 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 PDIUSB12*, 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.nxp.com.

1.4.3 Configuration Information

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 Wind River USB for VxWorks 6. For more detailed project facility, link-time, and run-time configuration information, see [2. Configuring and Building Wind River USB](#).

1.4.4 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.

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



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.

2

Configuring and Building Wind River USB

- 2.1 Introduction 11
- 2.2 Configuring and Building Wind River USB 11
- 2.3 Configuring VxWorks with Wind River USB 12
- 2.4 Building VxWorks with Wind River USB 22
- 2.5 Initializing USB Hardware 22

2.1 Introduction

This chapter describes how to build a VxWorks bootable image to include USB. The host stack enables VxWorks to use USB devices. The peripheral stack allows a Windows machine to treat a VxWorks machine as a USB device.

2.2 Configuring and Building Wind River USB

General instructions for building a product into VxWorks appear in your Platform getting started guide. USB is mostly precompiled, though some configlettes do

exist. For the most part, USB configures itself at run time rather than with compile-time macros.

2.3 Configuring VxWorks with Wind River USB

This section describes the components, component parameters, and initialization procedures for placing the USB host stack and USB peripheral stack on the VxWorks image.

2.3.1 USB Host Stack Components and Parameters

The host components include the following:

INCLUDE_USB

This is the USB host driver component, required by all systems using USB.

INCLUDE_USB_INIT

This is the initialization component for the USB host driver (USBD).

INCLUDE_EHCI

This is the EHCI host controller driver.

INCLUDE_EHCI_INIT

This is the initialization component for the EHCI host controller driver.

INCLUDE_EHCI_BUS

Registers the EHCI controller driver with VxBus.

INCLUDE_OHCI

This is the OHCI host controller driver.

INCLUDE_OHCI_INIT

This is the initialization component for the OHCI host controller driver.

INCLUDE_OHCI_BUS

Registers the OHCI controller driver with VxBus.

INCLUDE_UHCI

This is the UHCI host controller driver.

INCLUDE_UHCI_INIT

This is the initialization component for the UHCI host controller driver.

INCLUDE_UHCI_BUS

Registers the UHCI controller driver with VxBus.

INCLUDE_USB_KEYBOARD

This is the USB keyboard class driver.

INCLUDE_USB_KEYBOARD_INIT

This is the initialization component for the USB keyboard class driver.

INCLUDE_USB_MOUSE

This is the USB mouse class driver.

INCLUDE_USB_MOUSE_INIT

This is the initialization component for the USB mouse class driver.

INCLUDE_USB_PRINTER

This is the USB printer class driver.

INCLUDE_USB_PRINTER_INIT

This is the initialization component for the USB printer class driver.

INCLUDE_USB_SPEAKER

This is the USB speaker class driver.

INCLUDE_USB_SPEAKER_INIT

This is the initialization component for the USB speaker class driver.

INCLUDE_USB_MS_BULKONLY

This is the USB mass storage bulk-only class driver.

INCLUDE_USB_MS_BULKONLY_INIT

This is the initialization component for the USB mass storage bulk-only class driver. See [Parameters and Default Values](#), p. 14, for information about parameters and default values.

INCLUDE_USB_MS_CBI

This is the USB mass storage CBI class driver. See [Parameters and Default Values](#), p. 14, for information about parameters and default values.

INCLUDE_USB_MS_CBI_INIT

This is the initialization component for the USB mass storage CBI class driver.

INCLUDE_USB_PEGASUS_END

This is the USB Pegasus communication class driver.

INCLUDE_USB_PEGASUS_END_INIT

This is the initialization component for the USB Pegasus communication class driver. See [Parameters and Default Values](#), p. 14, for information about parameters and default values.

INCLUDE_USB_AUDIO_DEMO

This enables the USB Speaker audio demonstration configlet found in *installDir/target/config/comps/src/usrUsbAudioDemo.c*, which demonstrates the playing of audio files through the speakers.

INCLUDE_USB_HEADSET_DEMO

This enables the USB headset audio demonstration configlet, found in *installDir/target/src/config/comps/src/usbBrcmAudioDemo.c*, which demonstrates a feedback from the audio headset microphone to the speakers.

Parameters and Default Values

The host component parameters and default values include the following:

USB Mass Storage Bulk Only Initialization Configuration

The configuration parameters for the **INCLUDE_USB_MS_BULKONLY_INIT** component are the following:

BULK_MAX_DEVS

USB Bulk Maximum Drives specifies the maximum number of bulk device drives supported. The default is 2.

BULK_DRIVE_NAME

USB Bulk Drive Name specifies the drive name assigned to a USB bulk-only device. The default is *"/bd"*.

BULK_MAX_DRV_NAME_SZ

USB Bulk Device Name Size specifies the maximum size of each USB bulk device name. The default is 20.

USB_BULK_NON_REMOVABLE_DISK

The new file system does a status check on the mass storage disk before every read and write operation to determine whether the media are present. For USB flash disks that report themselves as removable media, this may be fine. However, for many USB mass storage disks that consist of nonremovable media, the status check before every read/write operation can cause unnecessary delays and may hamper the performance of the USB disk. To

obtain high performance rates, set `USB_BULK_NON_REMOVABLE_DISK` to `TRUE`. The default is `FALSE`.

USB Mass Storage Control-Bulk-Interrupt Configuration

The configuration parameters for the `INCLUDE_USB_MS_CBI` component are the following:

`CBI_DRIVE_NAME`

USB CBI Drive Name specifies the name of the drive assigned to a USB CBI device. The default is `"/cbid"`

`UFI_MAX_DEVS`

Maximum UFI Devices specifies the maximum number of CBI devices supported. The default is `2`.

`UFI_MAX_DRV_NAME_SZ`

USB UFI Device Name Size specifies the maximum size of each device name. The default is `20`.

USB Pegasus End Initialization Configuration

The configuration parameters for the `INCLUDE_USB_PEGASUS_END_INIT` component are the following:

`PEGASUS_MAX_DEVS`

USB Pegasus Device Maximum Number specifies the maximum number of supported Pegasus devices. The default is `1`.

`PEGASUS_IP_ADDRESS`

Pegasus IP Address specifies the IP address of a USB Pegasus device. The default is `{"90.0.0.3"}`.

`PEGASUS_NET_MASK`

Pegasus Net Mask specifies the USB Pegasus device net mask. The default is `{0xffffffff}`.

`PEGASUS_TARGET_NAME`

Pegasus Target Name specifies the target name of a USB Pegasus device. The default is `{"usbTarg0"}`.

This is an example of the code for the Pegasus default configuration parameters:

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

Keyboard Parameters

The configuration parameters for the **INCLUDE_USB_KEYBOARD_INIT** component are the following:

USB_KBD_QUEUE_SIZE

This is the maximum bytes of data that can be stored in the keyboard circular buffer. The default value is 8.

USB_MAX_KEYBOARDS

This is the maximum number of USB keyboards that can be attached.

ATTACH_USB_KEYBOARD_TO_SHELL

Setting this constant in the Workbench project facility attaches the USB keyboard to the target shell. This replaces the default console device.

Required Components

The components required for the USB host stack are the following:

- **INCLUDE_USB** is required for all USB support. This includes support for USB D.
- The component for your particular driver:
 - **INCLUDE_EHCI**
 - **INCLUDE_OHCI**
 - **INCLUDE_UHCI**

Selecting the **EHCI**, **OHCI**, or **UHCI** component includes modules for that type of host controller. All host controller components require that **INCLUDE_USB** also be selected. More than one host controller can be present in the image at once.

Optional Components

To add initialization at startup, include the host controller driver initialization components that match your host controller. If you are going to include such a component, you need the following:

- **INCLUDE_USB_INIT**
- The initialization component for your particular driver:
 - **INCLUDE_EHCI_INIT**
 - **INCLUDE_OHCI_INIT**

- `INCLUDE_UHCI_INIT`
- The registration of the driver with VxBus
 - `INCLUDE_EHCI_BUS`
 - `INCLUDE_OHCI_BUS`
 - `INCLUDE_UHCI_BUS`

You can optionally include any of the following USB device components:

- `INCLUDE_USB_KEYBOARD`
- `INCLUDE_USB_MOUSE`
- `INCLUDE_USB_PRINTER`
- `INCLUDE_USB_SPEAKER`
- `INCLUDE_USB_MS_BULKONLY`
- `INCLUDE_USB_MS_CBI`
- `INCLUDE_USB_PEGASUS_END`

To add initialization at startup for peripheral devices, include the appropriate components. These components require that the USB host stack be present on the VxWorks image. To include device initialization at system startup, select any of the USB peripheral device components, including the corresponding driver module as follows:

- `INCLUDE_USB_KEYBOARD_INIT`
- `INCLUDE_USB_MOUSE_INIT`
- `INCLUDE_USB_PRINTER_INIT`
- `INCLUDE_USB_SPEAKER_INIT`
- `INCLUDE_USB_MS_BULKONLY_INIT`
- `INCLUDE_USB_MS_CBI_INIT`
- `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.

To include support for the USB keyboard and mouse class driver, choose `INCLUDE_SELECT`.

For more information, see [Select Support on USB Keyboard and Mouse Class Drivers](#), p.19.

Component Dependencies

Certain devices do not handle mass storage reset properly. The **BULK_RESET_NOT_SUPPORTED** macro is defined by default.

Some components include compiler macros and flags that are defined in the makefile in the *installDir/target/src/drv/usb/target* directory. This process is documented in the Platform getting started guide. The USB peripheral stack includes the following configuration options for rebuilding source:

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

Managing Dependencies

This section discusses the general component dependencies for the USB host stack.

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 that 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:

- **INCLUDE_DOSFS**
- **INCLUDE_DOSFS_MAIN**
- **INCLUDE_DOSFS_CHKDSK**
- **INCLUDE_DOSFS_FMT**
- **INCLUDE_FS_MONITOR**
- **INCLUDE_ERF**
- **INCLUDE_XBD**
- **INCLUDE_DEVICE_MANAGER**
- **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 needed 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.

Select Support on USB Keyboard and Mouse Class Drivers

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



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

Speaker Demo

If you are using the `INCLUDE_USB_HEADSET_DEMO` configuration, do not define `INCLUDE_USBTOL` or `INCLUDE_USB_AUDIO_DEMO`. You will need `INCLUDE_USB_SPEAKER` and `INCLUDE_USB_SPEAKER_INIT` to complete the configuration. See [USB Audio Demo Initialization](#), p.26, and [USB Headset Demo](#), p.20.

If you are using the `INCLUDE_USB_AUDIO_DEMO` configuration, do not define `INCLUDE_USB_SPEAKER_INIT`. This is because the speaker audio demo will call the speaker initialization routine. For more information, see [Communication Class Device Initialization](#), p.26.

The .wav file can be played from a mass storage device. If the .wav file is being played through an ATA device, then define the `INCLUDE_ATA` macro.

USB Headset Demo

The USB host stack contains a sample application that demonstrates the use of a USB headset integrated with the host stack and the USB speaker class driver. Once the headset demo is initialized one of the following two tests can be invoked:

- The first test is invoked by issuing the command **usbBrcmAudioSpkrTest** and supplying a .wav file. The file play repeatedly through the headset speakers. For example ->**usbBrcmAudioSpkrTest "logon.wav"**
- The second test is a microphone loopback test where the microphone inputs echo to the headset speakers. This test is invoked from the command line by the command ->**usbBrcmAudioMicrophoneTest**. There are no parameters.

Both tests continue endlessly until terminated with **usbBrcmAudioTestStop()**.

2.3.2 USB Peripheral Stack Components and Parameters

INCLUDE_USB_TARG

This is the core component of the USB peripheral stack and must be included for all systems.

INCLUDE_USB_TARG_INIT

This is the initialization component for the core component for the USB peripheral stack.

INCLUDE_NET2280

This macro includes the Netchip NET2280 target controller driver.

INCLUDE_PHILIPS1582

This macro includes the Philips ISP1582 target controller driver.

INCLUDE_PDIUSB12

This macro includes the Philips PDIUSB 12 target controller driver.

INCLUDE_KBD_EMULATOR

This macro includes the keyboard emulator.

INCLUDE_KBD_EMULATOR_INIT

This macro initializes the keyboard emulator.

INCLUDE_MS_EMULATOR

This macro includes the mass storage emulator.

INCLUDE_MS_EMULATOR_INIT

This macro initializes the mass storage emulator.

INCLUDE_PRN_EMULATOR

This macro includes the printer emulator.

INCLUDE_PRN_EMULATOR_INIT

This macro initializes the printer emulator.

Required Components

Include the USB peripheral stack components. At a minimum, you must include:

- **INCLUDE_USB_TARG**

You must also include one of the following drivers:

- **INCLUDE_NET2280**
- **INCLUDE_PDIUSB12**
- **INCLUDE_PHILIPS1582**

Optional Components

You can include the following function drivers:

- **INCLUDE_KBD_EMULATOR**
- **INCLUDE_PRN_EMULATOR**
- **INCLUDE_MS_EMULATOR**
(This also requires file system components as described in [File System Components](#), p.18.)

The **Mass Storage Emulator** requires file system components.

You can include the following initialization components for the emulators:

- **INCLUDE_KBD_EMULATOR_INIT**
- **INCLUDE_PRN_EMULATOR_INIT**
- **INCLUDE_MS_EMULATOR_INIT**

USBTool Components and Parameters

INCLUDE_USBTOOL

This is the code exerciser, which performs all necessary USB driver initialization and can be used for debugging. If **INCLUDE_USBTOOL** is

included, all the initialization components (**INCLUDE_XXX_INIT**) must be undefined.

Optional Components

You can include the USB Testing Tool: **INCLUDE_USBTOOL**.

usbTool performs all necessary USB driver initialization and therefore cannot be used if any of the initialization components (**INCLUDE_XXX_INIT** macros) are included. See , p.24, for details.

2.4 Building VxWorks with Wind River USB

For information about building VxWorks with Wind River USB, including build options, image types, and so on, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.



NOTE: The hardware configuration routine calls BSP routines for USB. The BSP interface for USB is defined in the file *installDir/target/config/BSP/usbPciStub.c* .

2.5 Initializing USB Hardware

This section covers hardware initialization.

2.5.1 Initializing the USB Host Stack Hardware

This section provides initialization instructions for all USB host stack subcomponents including the host controller, device, and USB demo.

Startup Routines

This section provides a detailed description of the USB initialization process. The USB host stack initialization process includes the following three parts:

1. initializing the USBBD component and registering the USB hub bus type with VxBus
2. initializing the EHCI, OHCI, and UHCI host controllers
3. registering the respective bus controller drivers with VxBus

There are no more configlettes for host controller drivers. The host controllers are initialized by calling the init routines of the driver directly.

USBBD Initialization

To initialize the USBBD, 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

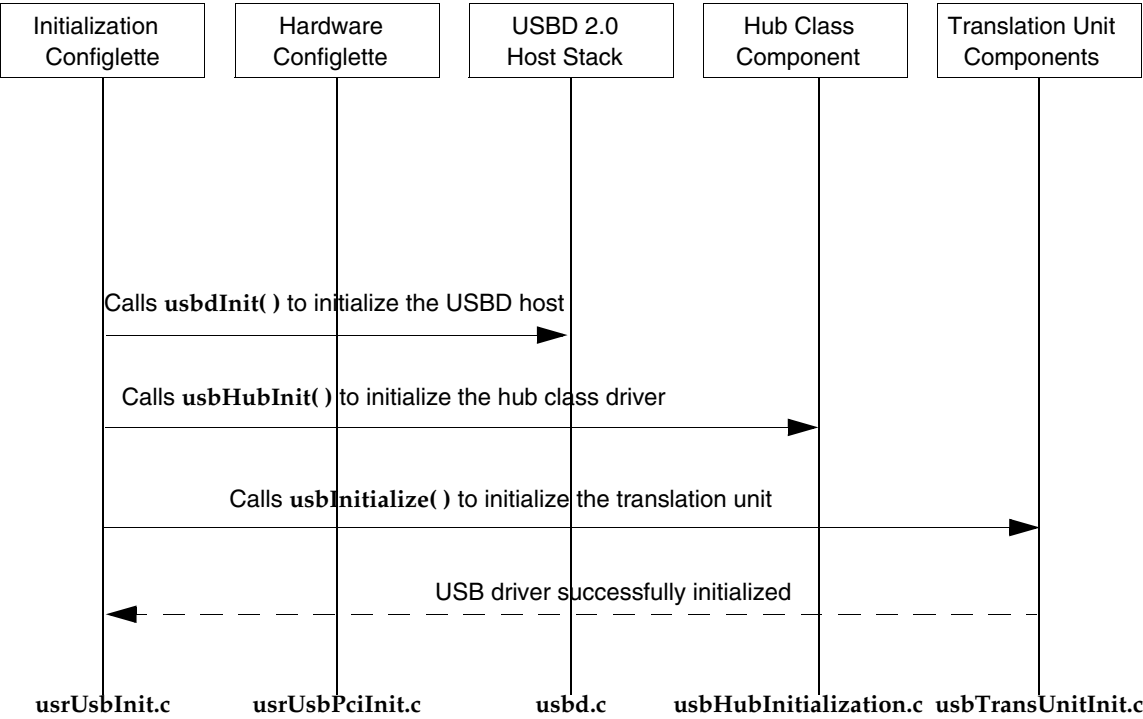
Once the USBBD is initialized, the host controllers can be initialized and registered with VxBus. When the controllers are registered with VxBus, VxBus will in turn announce them to the system and start them.

- **usbxhcdInit** (for example, **usbEhcdInit**) -- Initializes the host controller.
- **vxbUsbxhciRegister** (for example, **vxbUsbEhciRegister**) -- Registers the host controller with VxWorks.



NOTE: If the initialization components are included, the configlette routines are called to initialize the USBBD and host controllers during the bootup sequence itself.

Figure 2-1 **usbInit() Initialization Routine**



Initialization Dependencies

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



CAUTION: When you use the initialization components, you cannot use **usbTool**. The components are derived from **usbTool** itself, and cause compile-time errors when used 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()**:

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

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

```
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.



NOTE: See [4.7 Mass Storage Class Driver](#), p.86, for the possible values of MSC in this API name.

After a USB block device is created by means of **usbMSCBlkDevCreate()**, 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:

```
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:

USB SCSI_FLAG_READ_WRITE10	Use SCSI read/write ten.
USB SCSI_FLAG_READ_WRITE6	Use SCSI read/write six.

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**.

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

Communication Class Device Initialization

The USB host stack includes a configlet, 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 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.

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 audio class driver. This application was designed to run on a **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 and microphones. 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.

2.5.2 Initializing the USB Peripheral Stack Hardware

The USB peripheral stack contains configlet routines that are used to locate and configure resources required for initializing the USB peripheral hardware. These configlet routines are defined in *installDir/target/config/comp/src/usrUsbTargPciInit.c*. For more information, see [5.4 Enabling and Disabling the TCD](#), p.104.

The responsibilities of this configlet layer are:

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

Example 2-1 **NET2280 Configlet Routine**

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

```
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;
}
```


3

USB Host Drivers

- 3.1 Introduction 29
- 3.2 Architecture Overview 29
- 3.3 The USB Host Driver 34
- 3.4 Host Controller Drivers 54

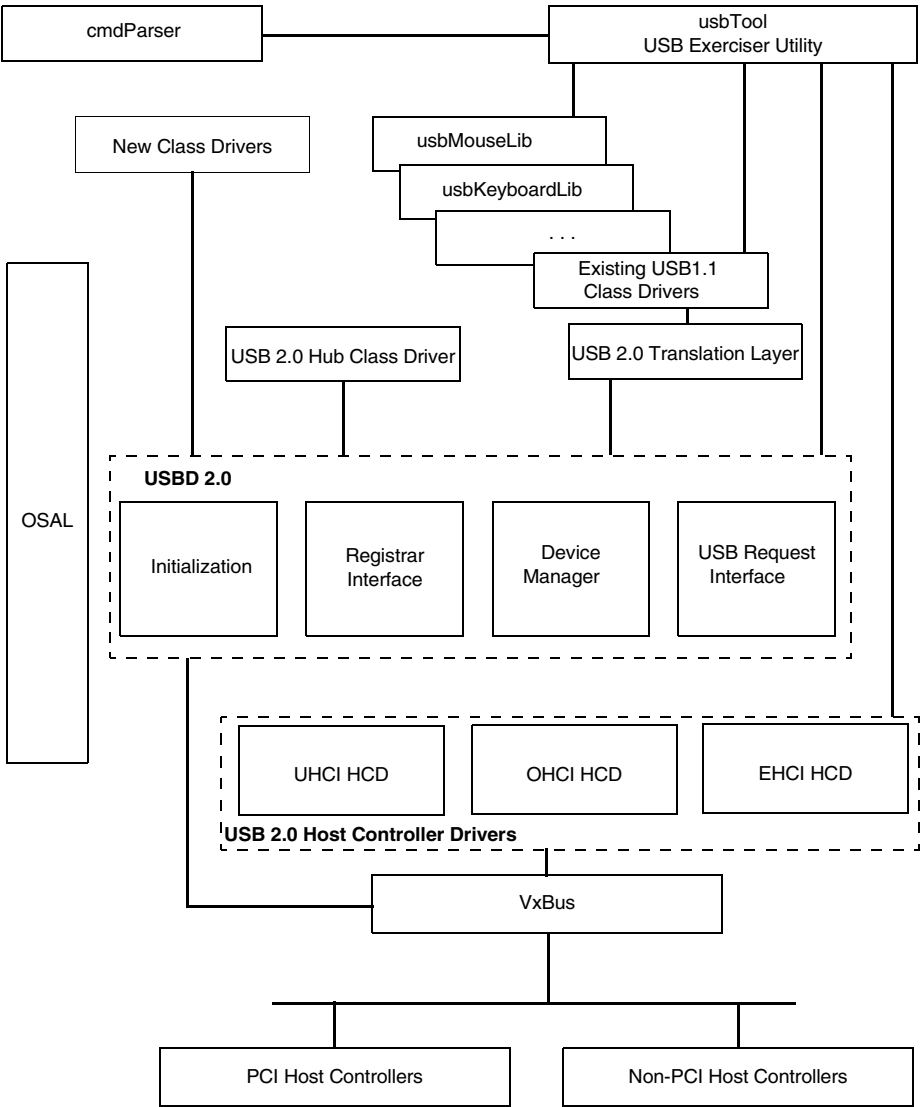
3.1 Introduction

This chapter provides a detailed overview of the USB host stack architecture as well as detailed usage information regarding the USB host driver (USBH) 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**



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.

3.2.1 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 prebuilt drivers in the following directories:

- `installDir/target/src/hwif/busCtrl/usb/hcd/ehcd` for EHCI HCs
- `installDir/target/src/hwif/busCtrl/usb/hcd/ohcd` for OHCI HCs
- `installDir/target/src/hwif/busCtrl/usb/hcd/uhcd` for UHCI HCs

The interface between the USBD and the HCD allows each HCD to control one or more underlying HCs. Also, Wind River's USBD can connect 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 HCDs. The USBD manages each USB peripheral device connected to the host and provides the path through which higher layers communicate with the USB.

Among its responsibilities, the USBD implements the USB protocol as explained in the USB 2.0 specification. It handles all the standard requests and routes other types of request 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 this driver include handling device connects, 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:

- `installDir/target/src/hwif/usb`
- `installDir/target/src/hwif/busCtrl/usb/hub`

3.2.2 Class Drivers

Figure 3-1 shows the USB class drivers at the top of the stack. USB class drivers are typical examples of client modules. USB class drivers are responsible for managing types of devices that can be connected to the USB; they rely on the USB D to provide the communication path to each device. Applications, diagnostics, and test programs are other examples of client modules that rely on the USB D 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:

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

3.2.3 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 routines, 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 *installDir/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 class driver for keyboard devices. For more information, see [4.3 Keyboard Driver](#), p.65.

usbMouseLib

This module is the class driver for mouse devices. For more information, see [4.4 Mouse Driver](#), p.73.

usbPrinterLib

This module is the class driver for printer devices. For more information, see [4.5 Printer Driver](#), p.77.

usbSpeakerLib

This module is the class driver for audio devices. For more information, see [4.6 Audio Driver](#), p.80.

usbBulkDevLib and **usbCbiUfiDevLib**

These modules are the class drivers for the mass storage class devices. For more information, see [4.7 Mass Storage Class Driver](#), p.86.

usbPegasusEndLib

These modules are the class drivers for the Ethernet networking control model communication class devices. For more information, see [4.8.1 Ethernet Networking Control Model Driver](#), p.92.

USB 2.0 Hub Class driver

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

USB 2.0 Translation Layer

This module preserves 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 USB host stack USBD and is composed of the following modules:

- The initialization module is responsible for initializing global data structures.
- The registrar provides an interface for class 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.
- The USB request interface allows 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.

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

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

3.3.1 USBDB 2.0

This section describes the USB host driver for USB 2.0.

Initializing the USBDB

When using the USBDB 2.0 interface, initializing the USBDB is a four-step process during the VxWorks boot-up process:

First, one or all the USB host controller drivers registers with VxBus, depending on the components included. This registration causes VxBus to discover the controller device and execute the appropriate VxBus initialization routines. This is done by calling **vxbUsbControllerRegister()**, where the controller is EHCI, OHCI or UHCI.

The EHCI, OCHI and UHCI drivers register themselves with VxBus for both PCI and local bus types.



NOTE: During boot-up, the function **hardwareInterfaceInit()** causes the registration of EHCI, OHCI and UHCI bus controller drivers.

Second, the USB D entry point **usbInit()**, must be called at least once. This routine initializes internal USB D data structures. In a given system, it is acceptable to call **usbInit()** once (for example, during the boot sequence) or many times (as during the initialization of each USB D client).

The third step is to call **usbHubInit()** to initialize the hub class driver.

Fourth, the appropriate host controller driver is registered with USB D by calling appropriate **usbHcdInit()** routine, where Hcd is Ehcd(EHCI HCD), Ohcd (OHCI HCD) or Uhcd (UHCI HCD). These routines perform some internal host controller-specific book-keeping, then registers the HCD with USB D.



NOTE: The second, third and fourth steps occur before the second level initialization of Vxbus takes place.

Order of Initialization

The **usbInit()** routine must be called before any other USB D routine, including **usbHubInit()**. However, not all USB D clients must call **usbInit()** before one or more HCDs have been attached to the USB D. Either of the following initialization sequence scenarios is acceptable:

Scenario #1: Traditional Boot Time Initialization

The order of initialization is as follows:

1. Each of the desired host controllers registers with VxBus.
2. Call **usbInit()**.
3. Call **usbHubInit()**.
4. Initialize the desired host controller driver and register the same with VxBus.
5. VxBus detects the host controllers and calls the initialization routines for each one. At the end of this phase, the host controller devices are up and ready for device detection.
6. Call the USB class driver initialization entry point.

Scenario #2: Hot Swap-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 [2.3 Configuring VxWorks with Wind River USB](#), p.12 to learn how to include these components.

2. The hot swap code calls the function **usbInit()**.
3. The hot swap code calls the function **usbOhciInit()**, **usbEhciInit()**, or **usbUhciInit()** to register the host controller driver with USB D.
4. The hot swap code calls the **vx bUs bEhciRegister()**, **vx bUs bOhciRegister()** or **vx bUs bUhciRegister()** function to register with VxBus.
5. If VxBus has detected the appropriate hardware, the appropriate host controller driver is then started.

Bus Tasks

For each host controller attached to the USB D, 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 USB D bus task has the VxWorks task name **BusM A**, **BusM B**, and so forth.

Registering Client Modules

The client module and the USB D layer share the **USBHST_DEVICE_DRIVER** data structure, defined in the file *installDir/target/h/usb/usbHst.h* as follows:

Example 3-1 USBHST_DEVICE_DRIVER Structure

```
/*  
 * 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;

    /* 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;
```

In order to communicate with USB devices, the client modules must register this data structure with the USBBD by calling **usbHstDriverRegister()**.

When a client registers with the USBBD, the USBBD allocates per-client data structures that are later used to track all requests made by that client.

There are two types of USB device:

- **Class-specific devices** - these devices adhere to the standard USB class specification supported by www.usb.org. Class-specific devices are identified by class, subclass, and protocol.
- **Vendor-specific devices** - these devices do not adhere to the standard USB class specification and their implementation are vendor-specific. Vendor-specific devices are identified by vendor ID, device ID, and binary-coded decimal (BCD) protocol.

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

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

Example 3-2 Registration and Deregistration of a USB Client

The following code fragment demonstrates the registration and deregistration of a USB 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, detachment, suspend, and resume, respectively.

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

USBHST_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;
```

```
/* register the client with USB D */
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 USB D 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.

```
USBHST_STATUS IssueDeviceSetDescriptor
(
    UINT32    hDevice,
    UINT32    uSize,
    PCHAR     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;
}
```

Data Transfer Interfaces

Once a client has configured a device, the client can begin to exchange data with that device using the data transfer interfaces provided by the USB D. 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 USB D 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,
    PCHAR     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_SINGALED);

/* 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;
}
```

3.3.2 USB 1.1 Compatibility

This is provided for the compatability of existing USB class drivers. Wind River recommends using the USB 2.0 API for all new development.

Registering Client Modules

Client modules that intend to make use of the USBDB to communicate with USB devices must, in addition to calling **usbdbInitialize()**, register with the USBDB by calling **usbdbClientRegister()**. When a client registers with the USBDB, the USBDB allocates per-client data structures that are later used to track all requests made by that client. During client registration, the USBDB also creates a callback task for each registered client (see *Client Callback Tasks*, p.43). After successfully registering a new client, the USBDB returns a handle, **USBDB_CLIENT_HANDLE**, that must be used by that client when making subsequent calls to the USBDB.

When a client no longer intends to use the USBDB, it must call **usbdbClientUnregister()** 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 Deregistration of the USBDB Client

The following code fragment demonstrates the registration and deregistration of a USBDB client:

```
USBDB_CLIENT_HANDLE usbdbClientHandle;

/* Register a client named "USBDB_TEST" with the USBDB. */
if (usbdbClientRegister ("USBDB_TEST", &usbdbClientHandle) != OK)
{
    /* Attempt to register a new client failed. */
    return ERROR;
}

/* Client is registered...application code follows. */
...
...
/* Unregister the client. */
usbdbClientUnregister (usbdbClientHandle);
```

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 USBDB clients are present, one client may interfere with the timely execution of other clients' service of time-sensitive USB traffic. The Wind River USBDB introduces per-client callback tasks to manage this problem.

Many USB events can result in callbacks to a USBDB client. For example, whenever the USBDB completes the execution of a USB I/O request packet (IRP), the client's

IRP callback routine is invoked. Similarly, whenever the USB D recognizes a dynamic attachment event, one or more client's dynamic attachment callback routines are invoked. Instead of invoking these callback routines immediately, the USB D schedules the callbacks to be performed by the callback task for the appropriate USB D clients. Normally, the callback task for each client is "dormant" (in a blocked state). When the USB D schedules a callback for a client, the corresponding client callback task "wakes up" (unblocks) and performs the callback. This approach allows the USB D 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 **usbClientRegister()**. 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 USB D or other USB D clients.

Client callback tasks have the VxWorks task name **tUsbdCln**.

Dynamic Attachment Registration

A typical USB D client wants to be notified whenever a device of a particular type is attached or removed. By calling the **usbDynamicAttachRegister()** 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 BCD device (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 BCD device (for vendor specific devices) when registering through **usbDynamicAttachRegister()**.

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** (the subclass and protocol are defined

in **usbHid.h**). In response, by means of the callback mechanism, the USBBD notifies the keyboard class driver whenever a device matching exactly this criterion is attached or removed.

In other cases, a client's interest is broader. In this case, the constant **USBBD_NOTIFY_ALL** (defined in **usbLib.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 **USBBD_NOTIFY_ALL**.

While a typical client makes only a single call to **usbDynamicAttachRegister()**, there is no limit to the number of concurrent notification requests a client can make.

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 demonstrate the correct use of the dynamic attachment registration routines:

```

/*****
 * attachCallback - called by USBBD when a device is attached/removed
 *
 * The USBBD invokes this callback when a USB device is attached to or
 * removed from the system. <nodeId> is the USBBD_NODE_ID of the node being
 * attached or removed. <attachAction> is USBBD_DYNA_ATTACH or
 * USBBD_DYNA_REMOVE.
 *
 * RETURNS: N/A
 */
LOCAL VOID attachCallback
(
    USBBD_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 USBBD_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. USBD_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, USBD_NOTIFY_ALL, FALSE, attachCallback) != OK)
{
    /* Attempt to register for dynamic attachment notification failed.*/
    return ERROR;
}

/*
 * attachCallback() - above - is now called whenever a USB printer
 * is attached to or removed from the system.
 */
...

/*
 * Cancel the dynamic attachment registration. Each parameter
 * must match exactly those found in an earlier call to
 * usbdDynamicAttachRegister().
 */
usbdDynamicAttachUnRegister (usbdClientHandle, USB_CLASS_PRINTER,
    USB_SUBCLASS_PRINTER, USBD_NOTIFY_ALL, attachCallback);
```

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

```
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 only for 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 **usbDynamicAttachRegister()**. The **deviceClass**, **deviceSubClass**, and **deviceProtocol** are set to **class**, **subclass**, and **protocol**, respectively.

Example 3-7 Registering a Callback

The following code fragment demonstrates the registration of the callback using **usbDynamicAttachRegister()** for class-specific and vendor-specific devices.

```
/* for class specific devices */
usbDynamicAttachRegister (usbClientHandle,
                          class,
                          subclass,
                          protocol,
                          FALSE,
                          callbackFunction
                          );

/* for vendor specific devices */
usbDynamicAttachRegister (usbClientHandle,
                          vendorID,
                          productID,
                          bcdDevice,
                          TRUE,
                          callbackFunction
                          );
```

Node IDs

USB devices are always identified using a **USB_NODE_ID**. The **USB_NODE_ID** is, in effect, a handle created by the translation unit to track a device (the node ID was created by the USB 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 USB can manage these details internally.

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

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 called bus enumeration routines, and they allow USB 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 in 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 USB 1.1 standard routines are as follows:

- **usbdFeatureClear()** - clears a USB feature
- **usbdFeatureSet()** - sets a USB feature
- **usbdConfigurtionGet()** - 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 USBDB itself takes care of certain USB configuration issues automatically. Most critically, the USBDB 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 USBDB to malfunction.

The USBDB also monitors configuration events. When a USBDB client invokes a USBDB routine that causes a configuration event, the USBDB automatically resets the USB data toggles (**DATA0** and **DATA1**) associated with the device pipes and endpoints. For example, a call to the USB command **usbdbConfigurationSet()** issues the set configuration command to the device, and in the process resets the data toggle to **DATA0**. Because the USBDB 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.51. For more information about USB data toggles, see the *USB Specification 2.0*.

Example 3-8 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:

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

UINT16 actLen;

/* USB_CONFIG_DESCRIPTOR defined in usb.h */
pUSB_CONFIG_DESCRIPTOR pCfgDescr;

/* USB_INTERFACE_DESCRIPTOR is also in usb.h */
pUSB_INTERFACE_DESCRIPTOR 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 (usbdbDescriptorGet (usbdbClientHandle, nodeId,
    USB_RT_STANDARD | USB_RT_DEVICE, USB_DESCRIPTOR_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.¹ The channel between a USB 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
- the transfer type
- the 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 USB client creates a **USBD_PIPE_HANDLE**, which the client must use for all subsequent operations on the pipe.

Example 3-9 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:

-
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.

```

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 (usbdbPipeCreate (usbdbClientHandle, 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. In order to destroy the pipe, one must specify the **USBD_CLIENT_HANDLE** (the handle to the client module) and **USBD_PIPE_HANDLE** (the handle to the pipe).

Example 3-10 Deleting a Pipe

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

```

USBD_PIPE_HANDLE outPipeHandle;
USBD_CLIENT_HANDLE usbdbClientHandle;

/* Destroys the pipe */

if (usbdbPipeDestroy (usbdbClientHandle, outPipeHandle) != OK)
{
    /* We failed to destroy the pipe. */
    return ERROR;
}

```



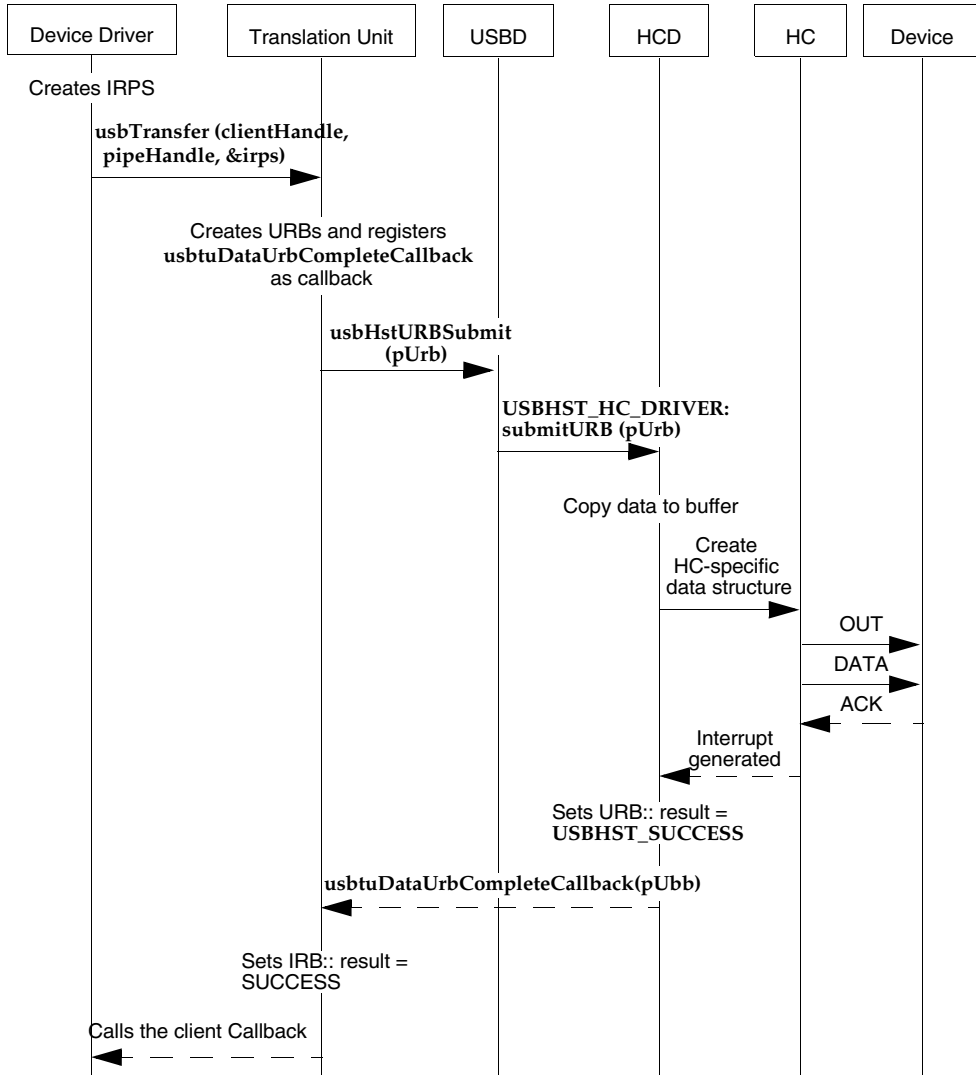
NOTE: When the **usbdbPipeDestroy()** routine is called, it looks for IRPs pending on the endpoint. If it finds any, it aborts all the pending transfers on the endpoint, then destroys the pipe.

Data Flow

Once a client has completely configured a pipe, it can begin to exchange data with the device using the pipe and transfer routines provided by the USB.

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.

Figure 3-2 State-Level Diagram of Data Flow



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

Example 3-11 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 (usbdtTransfer (usbdtClientHandle, outPipeHandle, &irp) != OK)
{
    /* An error here means that our IRP was malformed. */
    return ERROR;
}

```

The **usbdtTransfer()** function returns as soon as the IRP has been successfully queued. If there is a failure in delivering the IRP to the HCD, then **usbdtTransfer()** returns an error. The actual result of the IRP is 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 the **usbdtTransferAbort()** routine to cancel all submitted IRPs.

3.4 Host Controller Drivers

This section describes the interface and requirements for 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 USB D data structures and USB D interfaces used by the host controller driver.

3.4.1 Registering the Host Controller Driver

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

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

3.4.2 USBHST_HC_DRIVER Structure

The **USBHST_HC_DRIVER** structure contains the HCD function pointers. During HCD initialization, this structure is populated and registered with the USB host stack. The USB D uses the function pointers in this structure to communicate with the HCD

Example 3-12 USBHST_HC_DRIVER Structure

This structure takes the following form:

```
/* 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);

/* 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)
    (UINT8      uRelativeBusIndex,
     void *      pContext,
     USBHST_STATUS nStatus);

```

```
/* Function pointer to submit the status of the reset TT request */  
  
USBHST_STATUS (*resetTTRequestComplete) (UINT8      uRelativeBusIndex,  
                                          void *      pContext,  
                                          USBHST_STATUS nStatus);  
  
} USBHST_HC_DRIVER, *pUSBHST_HC_DRIVER;
```

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 the 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 the given endpoint, deletes it and updates the list. 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 for requests pending for an endpoint and returns a value accordingly. If there are pending requests, it returns a Success status.

getFrameNumber()

This routine obtains the current frame number from the frame number register.

setBitRate()

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

clearTTRequestComplete() and **resetTTRequestComplete()**

These routines are used to handle errors that occur during split transactions. These routines are implemented only for an EHCI host controller. The EHCD registers these routines with the USBBD 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 *installDir/target/h/usb/usbHst.h*.

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

    /*
     * 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 the HCD by the USBD.

```

/* Register the HCD with the USBD */

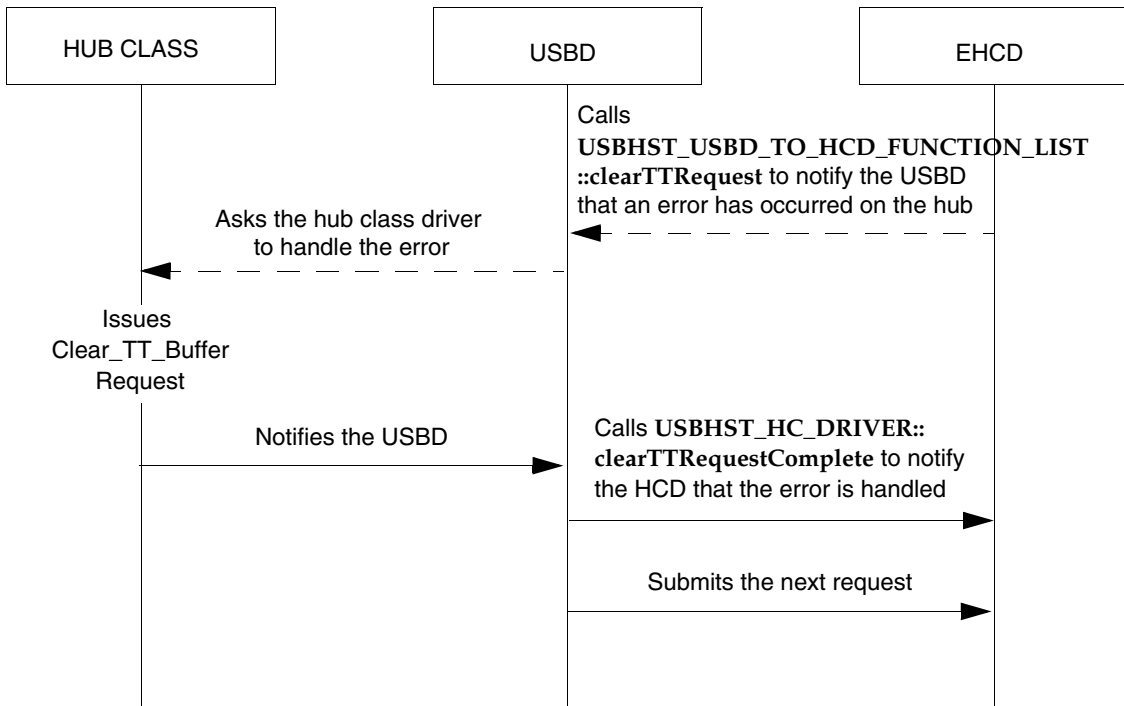
Status = usbHstHCDRegister(g_pEHCDriverInfo, &g_EHCDHandle,
                           &g_USBDRequestFunctions);

```

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 transaction translator (TT) buffers are cleared.

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

Figure 3-3 Handling Split Errors





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, the following occurs:

1. The `usbHstHCDRegister()` routine is called once to register the OHCI HCD.
2. The `usbHstBusRegister()` routine is called three times to register the three OHCI host controllers.

3.4.5 Deregistering the Bus for the Host Controller

All HCDs deregister 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 deregistered.

3.4.6 Deregistering the Host Controller Driver

All HCDs deregister themselves from the USB host stack by calling the `usbHstHCDDeregister()` routine. If there are buses registered for the HCD, the HCD is not deregistered.

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 it returns 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`

- USBHST_STALL_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 USB device driver interrogates the root hub, just as it would other hubs, by issuing USB **GET_DESCRIPTOR** requests. If the hub class driver is registered, the USB device driver calls it 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 61
- 4.2 Hub Class Driver 62
- 4.3 Keyboard Driver 65
- 4.4 Mouse Driver 73
- 4.5 Printer Driver 77
- 4.6 Audio Driver 80
- 4.7 Mass Storage Class Driver 86
- 4.8 Communication Class Drivers 92

4.1 Introduction

This chapter provides information on the USB class drivers that are provided with the Wind River USB host stack. prebuilt class drivers for several USB device types are included in your installation.

4.2 Hub Class Driver

Hubs are a class of USB device 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 USB D 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 USB D by calling **usbHstDriverRegister()**. By registering with the USB D, the hub class driver provides an interface to the USB D for adding, removing, suspending, and resuming a device. [Figure 4-1](#) illustrates the registration process.

Figure 4-1 Registering the Hub Class Driver

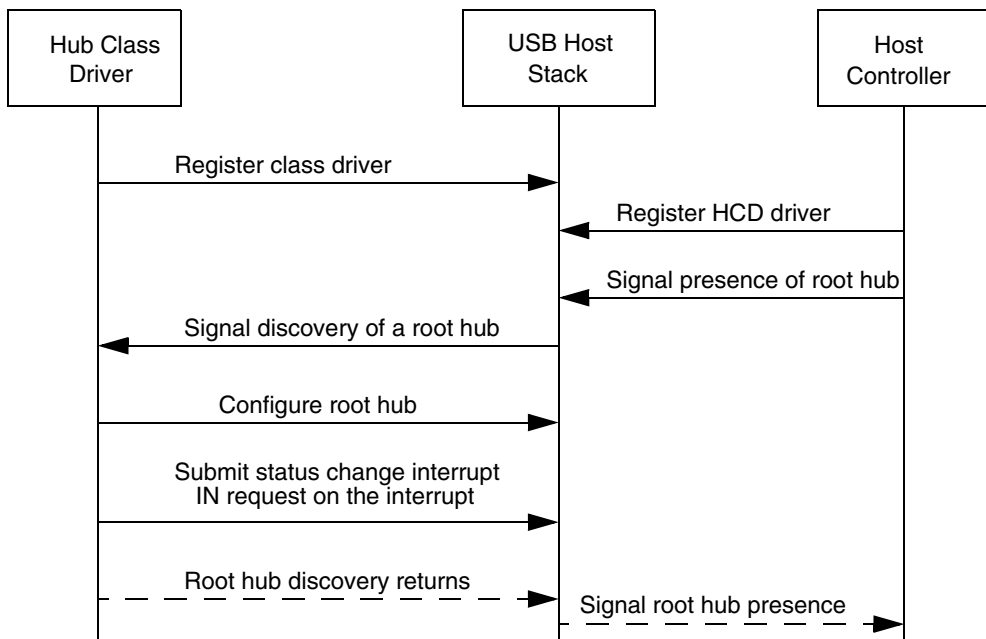
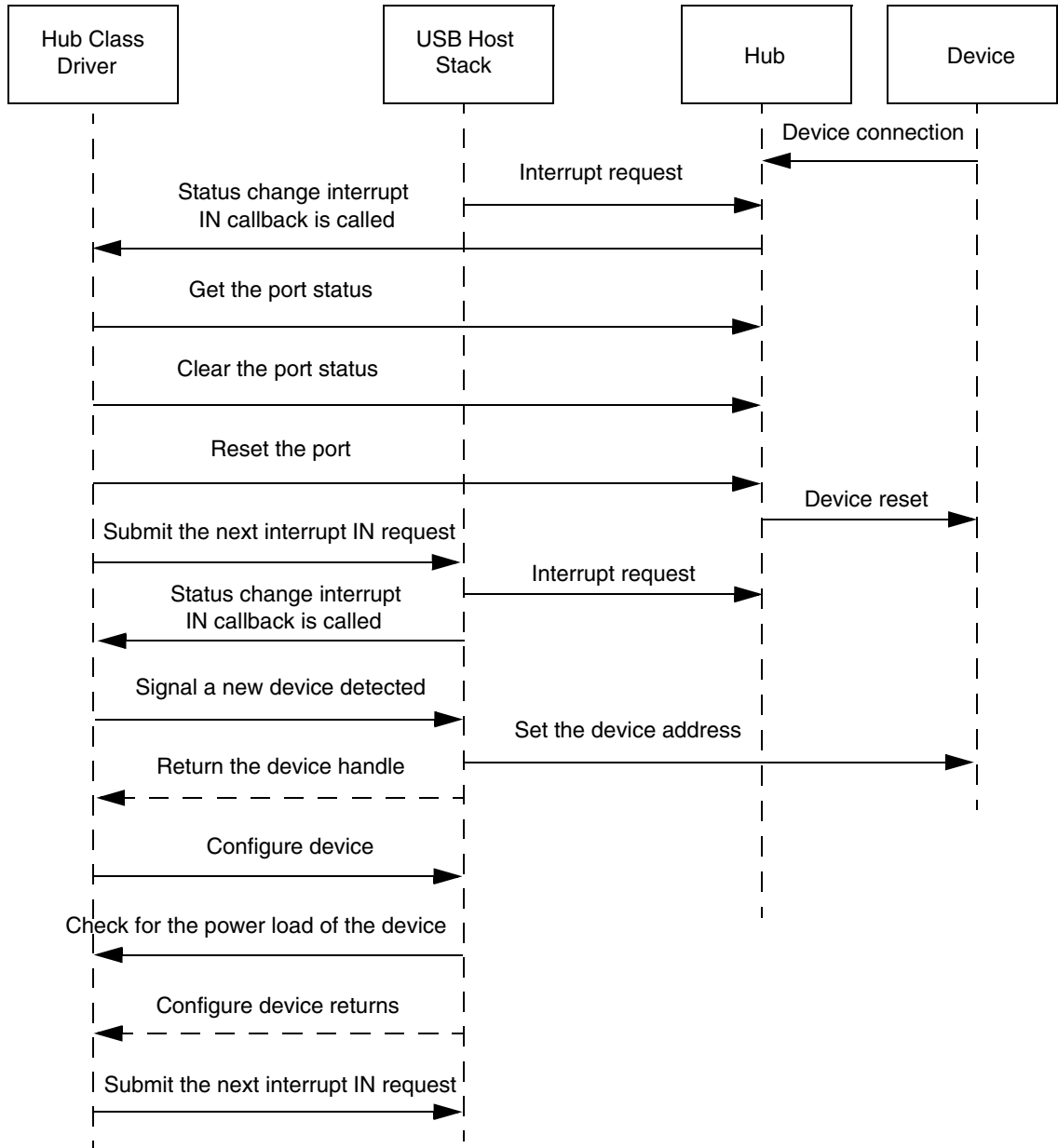


Figure 4-2 Connecting a Device to a Hub



After registering itself with the USBD, the hub class driver configures the root hub for all of the host controller drivers registered with the USBD. Then the hub class driver starts polling the interrupt pipe of the root hub to monitor changes on the downstream ports.

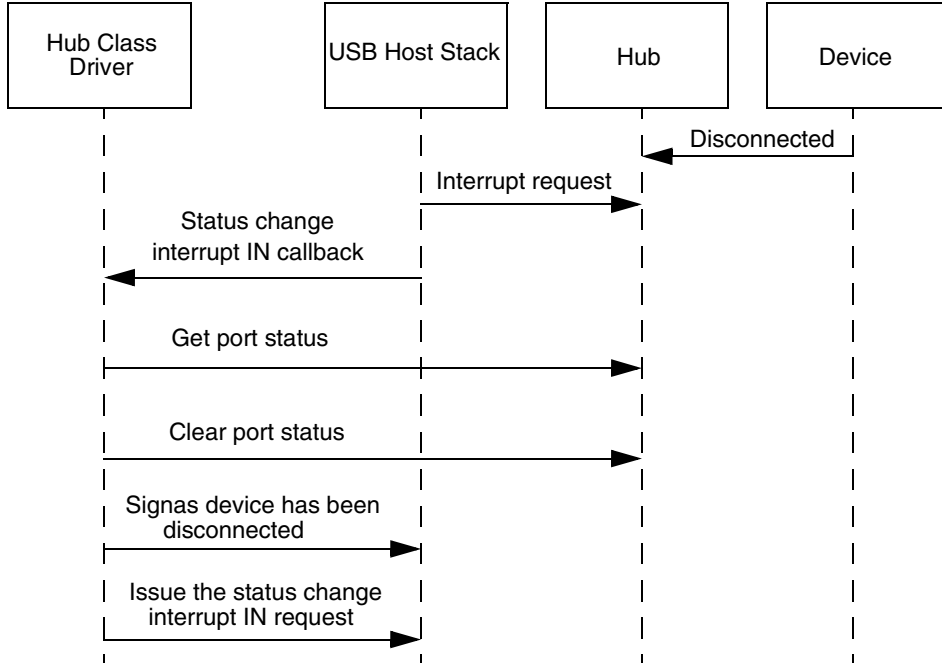
4.2.2 Connecting a Device to a Hub

When a device is connected to a hub, the USB host stack informs the hub class driver. The interaction shown in [Figure 4-2](#) occurs between the hub class driver, the USB host stack, the hub, and the device.

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.

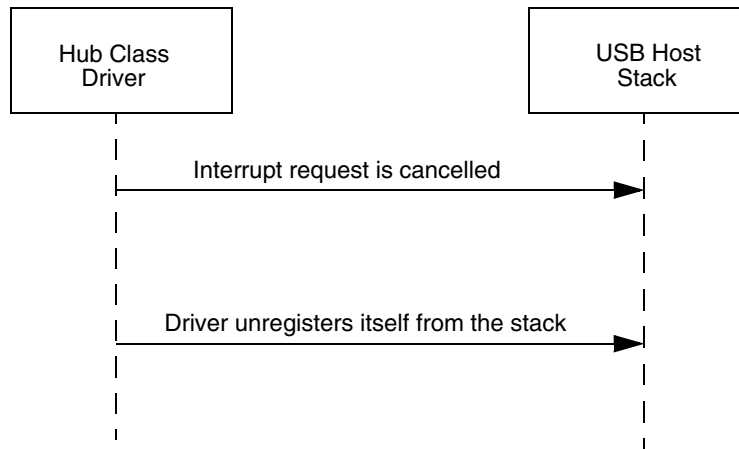
Figure 4-3 Removing a Device from a Hub



4.2.4 Deregistering the Hub Class Driver

The hub class driver deregisters with 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 deregistration process.

Figure 4-4 Deregistering the Hub Class Driver



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 HID.



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 to 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 USBSD has been properly initialized. Please refer to [2.5.1 Initializing the USB Host Stack Hardware](#), p.22, for information on how to initialize the 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 registration with the lower layer happens on the very first call to the **usbKeyboardDevInit()**. This call increments the usage counter from zero to one. For all subsequent calls to the initialization routine, the usage counter is simply incremented.

The following code example demonstrates how this is accomplished:

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

STATUS usbKeyboardDevInit (void)
{
    /* If not already initialized, then initialize
     * internal structures and connection to USBSD.
     */

    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 USB D by calling **usbClientRegister()**. 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 **USB_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 layers by calling the routine **usbDynamicAttachRegister()**. This routine is called whenever a device of a particular class, subclass, and protocol is connected to or disconnected from the USB subsystem. The callback routine takes the following form:

```
typedef VOID (*USB_ATTACH_CALLBACK)
(
    USB_NODE_ID nodeId,      /* USB 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 of **usbDynamicAttachRegister()**.

Example 4-1 Registering the Keyboard Class Driver

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

```
/* Registering the Keyboard Class Driver */

STATUS usbKeyboardDevInit (void)
{
    .....
    .....

    if (usbClientRegister (KBD_CLIENT_NAME, &usbHandle) != OK
        || usbDynamicAttachRegister (usbHandle,
                                     USB_CLASS_HID,
                                     USB_SUBCLASS_HID_BOOT,
                                     USB_PROTOCOL_HID_BOOT_KEYBOARD, FALSE,
                                     usbKeyboardAttachCallback)
```

```
!= OK)

return ERROR;

.....
}
```

The keyboard class driver unregisters itself with the lower layers by calling the routine **usbdClientUnregister()**.

```
/*
 * un-registering the keyboard class driver
 */

usbdClientUnregister (usbdHandle);
usbdHandle = NULL;
```

4.3.4 Dynamic Device Attachment

Unlike most SIO drivers, the **usbKeyboardLib** driver does not support a fixed number of channels. Rather, USB keyboards can be added to 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:

```
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, **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 4-2 **SIO_CHAN Locking**

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

```
/* 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.
 */
...
/* Unregister for keyboard notification and shut down usbKeyboardLib. */
usbKeyboardDynamicAttachUnRegister (kbdAttachCallback, (pVOID) 1234);
usbKeyboardDevShutdown ();
```

Example 4-3 **Keyboard Attachment**

The keyboard attachment callback might resemble the following:

```

/*****
 * 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 into 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 USB and USB HCD always operate in interrupt mode.

Example 4-4 Keyboard to Display Keystrokes

The following code fragment demonstrates the use of **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 key press. If no report is received within a preset 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 preset rate. In the current implementation, the preset interval (the delay) is 0.5 seconds, and the repeat rate is 15 characters per second.

4.3.8 Uninitializing the Keyboard Class Driver

The client application uninitializes the keyboard class driver by calling the routine **usbKeyboardDevShutdown()**. This routine uninitializes 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 uninitialization of a driver happens only when the usage count becomes zero. Otherwise, for every call to **usbKeyboardDevShutdown()**, it is decremented by one.

4.4 Mouse Driver

USB mice are described in 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 HID devices, 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 USBDB and other internal resources needed for operation. All interaction with the USB host controllers and devices is handled through the USBDB.

Unlike some SIO drivers, **usbMouseLib** does not include data structures that require initialization before calling **usbMouseDevInit()**.

However, before a call to **usbMouseDevInit()**, the caller must ensure that the USBDB has been properly initialized by calling, at a minimum, **usbdbInitialize()**. The caller must also confirm that at least one USB HCD is attached to the USBDB, using **usbdbHcdAttach()**, before mouse operation can begin; however, it is not necessary to call **usbdbHcdAttach()** before initializing **usbMouseLib**. The **usbMouseLib** driver uses the USBDB 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 USBDB and USB HCDs, 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 USB D by calling **usbClientRegister()**. 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 **USB_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 layers by calling the routine **usbDynamicAttachRegister()**. This routine is called whenever a device of a particular class, subclass, and protocol is connected to or disconnected from the USB subsystem. The callback routine takes the following form:

```
typedef VOID (*USB_ATTACH_CALLBACK)
(
    USB_NODE_ID nodeId,      /* USB 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 of **usbDynamicAttachRegister()**.

Example 4-5 Registering the Mouse Class Driver

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

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

STATUS usbMouseDevInit (void)
{
    .....
    .....

    if (usbClientRegister (KBD_CLIENT_NAME, &usbHandle) != OK
        || usbDynamicAttachRegister (usbHandle,
                                     USB_CLASS_HID,
                                     USB_SUBCLASS_HID_BOOT,
                                     USB_PROTOCOL_HID_BOOT_KEYBOARD, FALSE,
                                     usbMouseAttachCallback)
```

```

                                != OK)

    return ERROR;

    .....
}

```

The mouse class driver unregisters itself with the lower layers by calling the routine **usbClientUnregister()**.

```

/* un-registering the mouse class driver */

usbClientUnregister (usbHandle);
usbHandle = 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:

```

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 4-2 in 4.3.4 Dynamic Device Attachment](#), p.68.

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**:

```
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:

```
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 may 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 Uninitializing the Mouse Class Driver

The client application uninitializes the mouse class driver by calling the routine **usbMouseDevShutdown()**. This routine uninitializes the internal data structures and removes the mouse class driver from the USB subsystem.

4.5 Printer Driver

USB printers are described in HID-related specifications. The printer class driver specification presents two kinds of printers: unidirectional printers (output only) and bidirectional printers (capable of both output and input). The **usbPrinterLib** driver is capable of handling both types. If a printer is unidirectional, the driver only allows characters to be written to the printer; if the printer is bidirectional, 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 USBDB and other internal resources needed for operation. All interaction with the USB host controllers and devices is handled through the USBDB.

Unlike some SIO drivers, **usbPrinterLib** does not include data structures that require initialization before calling **usbPrinterDevInit()**.

However, before a call to **usbPrinterDevInit()**, the caller must ensure that the USBDB has been properly initialized. For details for how to initialize the USB host stack, refer to [2.5.1 *Initializing the USB Host Stack Hardware*](#), p.22.

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 USBDB and USB HCDs, 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 USB D by calling **usbClientRegister()**. 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, it registers a callback routine with the lower layers by calling the routine **usbDynamicAttachRegister()**. This routine is called whenever a device of a particular class, subclass, and protocol is connected to or disconnected from the USB subsystem. The callback routine takes the following form:

```
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 of **usbDynamicAttachRegister()**.

Example 4-6 Registering the Printer Class Driver

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

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

STATUS usbPrinterDevInit (void)
{
    .....
    .....
    if (usbClientRegister (PRN_CLIENT_NAME, &usbHandle) != OK
        || usbDynamicAttachRegister (usbHandle,
                                    USB_CLASS_PRINTER,
                                    USB_SUBCLASS_PRINTER,
                                    USBD_NOTIFY_ALL, FALSE,
                                    usbPrinterAttachCallback)
```

```

                                     != OK)
    3
    return ERROR;
    .....
}

```

The printer class driver unregisters itself with the lower layers by calling the routine **usbClientUnregister()**.

```

/* un-registering the printer class driver */

usbClientUnregister (usbHandle);
usbHandle = NULL;

```

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

As with **usbKeyboardLib**, **usbPrinterLib** maintains a usage count for each **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, **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 4-2](#) in *4.3.4 Dynamic Device Attachment*, p.68.

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 allow the caller to retrieve the USB printer's device ID string, its type (unidirectional or bidirectional) and 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

usbPrinterLib sets up a pipe to output bulk data to each USB printer connected to the system. This is the pipe through which printer control and page description data are sent to the printer. If the printer is bidirectional, **usbPrinterLib** also sets up a pipe to receive bulk input data from the printer. The meaning of the data received from a bidirectional printer depends on the 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 the polled mode returns an error.

4.6 Audio Driver

USB speakers and microphones are described in *USB Device Class Definition for Audio Devices*. The Wind River implementation of the USB audio driver, **usbSpeakerLib**, supports only USB audio devices as defined by this specification

and ignores other types of USB audio devices, such as MPEG and MIDI devices. For USB 2.4, Wind River enhanced **usbSpeakerLib** to support the use of input audio devices such as USB microphones.



NOTE: Some models of USB audio devices 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 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 [4.3.5 ioctl Routines](#), p.71).

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 audio devices. As with other VxWorks USB class drivers, the standard driver interface has been expanded to support features unique to USB and to audio devices in general. Routines have been added to allow callers to recognize the dynamic attachment and removal of audio devices and ioctl routines have been added to retrieve and control additional settings related to audio device operation. The **SEQ_DEV** interface has been enhanced to support headset operation. Microphone support has been added. Previously, **SEQ_DEV** supported only speakers and the nomenclature reflects this by often using the word “speaker” when, strictly speaking, it should now say “audio.” There are no plans to change this, however.

4.6.2 Initializing the Audio 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 USB and other internal resources needed for operation.

However, before a call to **usbSpeakerDevInit()**, the caller must ensure that the USB has been properly initialized. Please refer to [2.5.1 Initializing the USB Host Stack Hardware](#), p.22 for instructions on how to initialize USB host stack.

The USB speaker library uses the USB dynamic attachment services to recognize USB audio device attachment and detachment on the fly.

4.6.3 Registering the Audio Driver

The audio class driver registers with the USB dynamic attachment services by calling **usbClientRegister()**. In response to this call, the translation unit allocates audio device client data structures and a audio device client callback task.

Once the audio 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 audio class driver is successfully registered, the driver registers a callback routine with the lower layers by calling the routine **usbDynamicAttachRegister()**. This routine is called whenever a device of a particular class, subclass, and protocol is connected to or disconnected from the USB subsystem. The callback routine takes the following form:

```
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 of **usbDynamicAttachRegister()**.

Example 4-7 Registering the Audio Class Driver

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

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

STATUS usbSpeakerDevInit (void)
{
    .....
    .....
    if (usbClientRegister (SPKR_CLIENT_NAME, &usbHandle) != OK
        || usbDynamicAttachRegister (usbHandle,
```

```

USB_CLASS_PRINTER,
USBD_NOTIFY_ALL,
USBD_NOTIFY_ALL, FALSE,
usbSpeakerAttachCallback)
!= OK)

return ERROR;
.....
}

```

The audio class driver unregisters itself with the lower layers by calling the routine **usbClientUnregister()**.

```

/* un-registering the speaker class driver */

usbClientUnregister (usbHandle);
usbHandle = NULL;

```

4.6.4 Dynamic Device Attachment

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

```

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 audio device 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 audio device 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 audio device is removed from the system when its usage count is zero, **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

As noted earlier, the operation of USB speakers is defined in the *USB Audio Class Specification*. Speakers, loosely defined, are those USB audio devices which provide an output terminal. For each USB audio device, **usbSpeakerLib** examines the descriptors which enumerate the units and terminals contained within the device. These descriptors define which kinds of units and terminals are present and how they are connected.

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

After determining that the audio device contains an output terminal, **usbSpeakerLib** builds a list of the audio formats supported by the device. **usbSpeakerLib** supports only audio streaming interfaces. No MIDI streaming is supported.

usbSpeakerLib creates a **SEQ_DEV** structure to control each USB speaker attached to the system and properly recognized by **usbSpeakerLib**. Each speaker is uniquely identified by the pointer to its corresponding **SEQ_DEV** structure.

Dynamic Attachment and Removal of Speakers

As with other USB devices, USB speakers may be attached or detached dynamically. **usbSpeakerLib** uses the dynamic attach services of the USB D in order to recognize these events. Callers of **usbSpeakerLib** may, in turn, register with **usbSpeakerLib** for notification when USB speakers are attached or removed using the **usbSpeakerDynamicAttachRegister()** function. When a USB speaker is attached or removed, **usbSpeakerLib** invokes the attach notification callbacks for all registered callers. The callback is passed the pointer to the affected **SEQ_DEV** structure and a code indicates whether the speaker is being attached or removed.

usbSpeakerLib maintains a usage count for each `SEQ_DEV` structure. Callers can increment the usage count by calling **usbSpeakerSeqDevLock()** and can decrement the usage count by calling **usbSpeakerSeqDevUnlock()**. When a USB audio device is removed from the system and its usage count is zero, **usbSpeakerLib** automatically removes all data structures, including the `SEQ_DEV` structure itself, that were allocated on behalf of the device. Sometimes, however, callers 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 functions provide a mechanism for callers to protect these data structures as needed.

Data Flow

Before sending audio data to a speaker device, the caller must specify the data format, such as PCM or MPEG, using an **ioctl()** (see below). The USB speaker itself must support the indicated 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 to flow continuously, **usbSpeakerLib** internally uses a double-buffering scheme. When the caller presents data to the **sd_seqWrt()** routine of the **usbSpeakerLib**, **usbSpeakerLib** copies the data into an internal buffer and releases the buffer of the caller. The caller passes the next buffer to **usbSpeakerLib**. When **usbSpeakerLib**'s internal buffer is filled, it blocks the caller until it can accept the new data. In this manner, the caller and **usbSpeakerLib** work together to ensure an adequate supply of audio data for an uninterrupted isochronous transmission.

Audio play begins after **usbSpeakerLib** has accepted half a second of audio data or when the caller closes the audio stream, whichever happens first. The caller must use the **ioctl()**s to open and close each audio stream. **usbSpeakerLib** relies on these **ioctl()**s to manage its internal buffers.

4.6.6 Recognizing and Handling USB Microphones

As with USB speakers, microphones may be attached or detached dynamically. **usbSpeakerLib** uses the USB's dynamic attach services in order to recognize these events. When a USB microphone is attached or removed, **usbSpeakerLib** invokes the attach notification callbacks for all registered callers. The callback is passed the pointer to the affected `SEQ_DEV` structure and a code indicates whether the microphone is being attached or removed.

Data Flow

When connecting a microphone, the caller must select from the formats available on the microphone the one appropriate for the desired application. Then, using `ioctl()`s, the caller specifies the format and the interval in which the caller obtains the data from `usbSpeakerLib`. `usbSpeakerLib`, then allocates an appropriate buffer size and posts isochronous IN requests to the USB microphone, filling the buffer as the IN requests complete.

The caller then posts an `sd_seqRd` to `usbSpeakerLib`, at appropriate intervals, to obtain the audio for further processing. Note that the reader does not block waiting for data, but returns all the data up to the requested buffer size. The caller always checks the returned value of the read for the amount of data actually read.

As with the USB speakers, double buffering provides a continuous stream of data; however, if the caller cannot service the data at a sustainable rate, the new data may overwrite the old in the data buffers.

A demonstration program using a USB headset (with a speaker and microphone) is provided in source form as a configlet. See [INCLUDE_USB_HEADSET_DEMO](#), p. 14.

4.7 Mass Storage Class Driver

USB mass storage class devices are described in the *Universal Serial Bus Mass Storage Class Specification Overview* 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 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*.



NOTE: *SCSI Primary Commands: 2 (SPC-2)* is available from Global Engineering, (800-854-7179). The *UFI Command Specification* is available on the Web (http://www.usb.org/developers/devclass_docs/usbmass-ufi10.pdf).

The subclass code of a device, 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 4-1 Device Subclass Codes and Corresponding Command Sets

Subclass Code	Command Block Specification	Comments
01h	Reduced Block Commands (RBC) T10 Project 1240-D	Typically, a flash device uses RBCs. However, any mass storage device can use RBCs.
02h	SFF-8020i or MMC-2 (ATAPI)	Typically, a CD/DVD device uses SFF-8020i or MMC-2 command blocks for its mass storage interface.
03h	QIC-157	Typically, a tape device uses QIC-157 command blocks.
04h	UFI	Typically, a floppy disk drive (FDD) device uses UFI command blocks.
05h	SFF-8070i	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.
06h	SCSI transparent command set	
07h-FFh	Reserved for future use	

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 with drivers and higher levels of functionality. An XBD structure is allocated by the driver that needs to use that interface.

When a mass storage block device insertion takes place, `usbMSCBlkDevCreate()` allocates an XBD for it. An insertion event is generated for the ERF which propagates that insertion event to any higher level function that may 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 `installDir/target/h/drv/xbd/xbd.h`.

- **BIO structure**

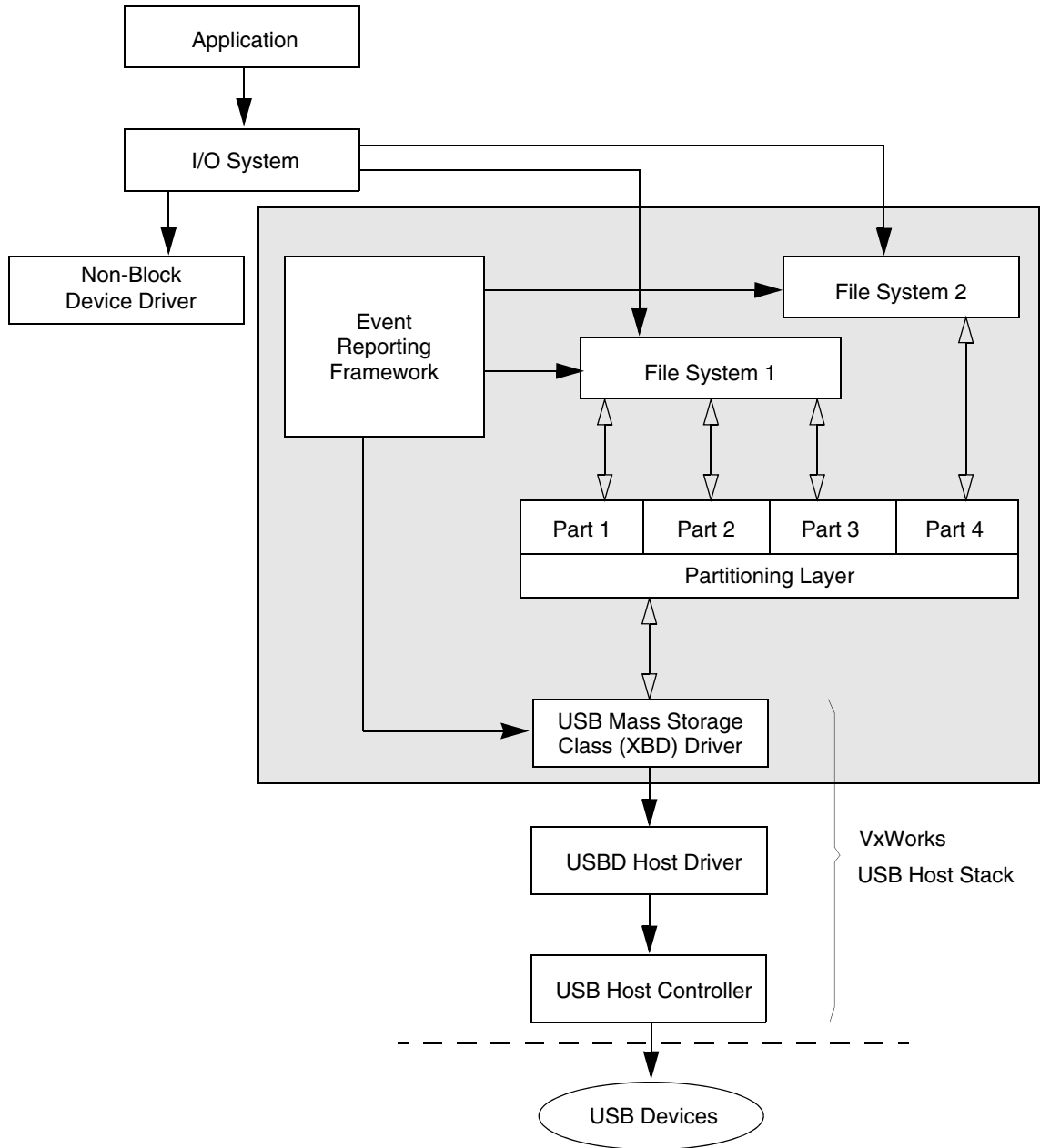
The **Block_IO** (BIO) structure contains the information necessary to read or write to the block device or to another XBD. The BIO structure is defined in `installDir/target/h/drv/xbd/bio.h`.

For each USB device, a `UsbMSCXbdStrategy` 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 logical unit (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. The device driver notifies the ERF of the insertion and creates an XBD interface for the mass storage device.
3. The ERF reports the insertion to the registered file systems. 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 disk partitions.

Figure 4-5 **USB Block Driver Hierarchy in a VxWorks System**



4.7.2 API Routines

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

usbMSCDevInit()

This is a general initialization routine. It performs all operations that are to be done one time only. It initializes the required data structures and registers the mass storage class driver with the USB D. 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 configurations.

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 hot swap 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 or removed, all clients are notified through the USBBD'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 deregistering a block device driver.

The mass storage class driver maintains a usage count of **XBD** structures. When a client uses an **XBD** structure, it informs the driver by calling **usbMSCDevLock()**. The driver increments the usage count for each **usbMSCDevLock()** call. When a client is finished with the **XBD** structure, it notifies 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, 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 **usbMSCDevInit()** routine initializes the mass storage class driver. This API call, in turn, initializes internal resources needed for its operation and registers a callback routine with the USBBD. The callback routine is then invoked whenever a USB mass storage class device is attached or removed.

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

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 only 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 I/O request packet (IRP) mechanism. This allows the user to specify a routine, `usbMSCIrpcCallback()`, 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.

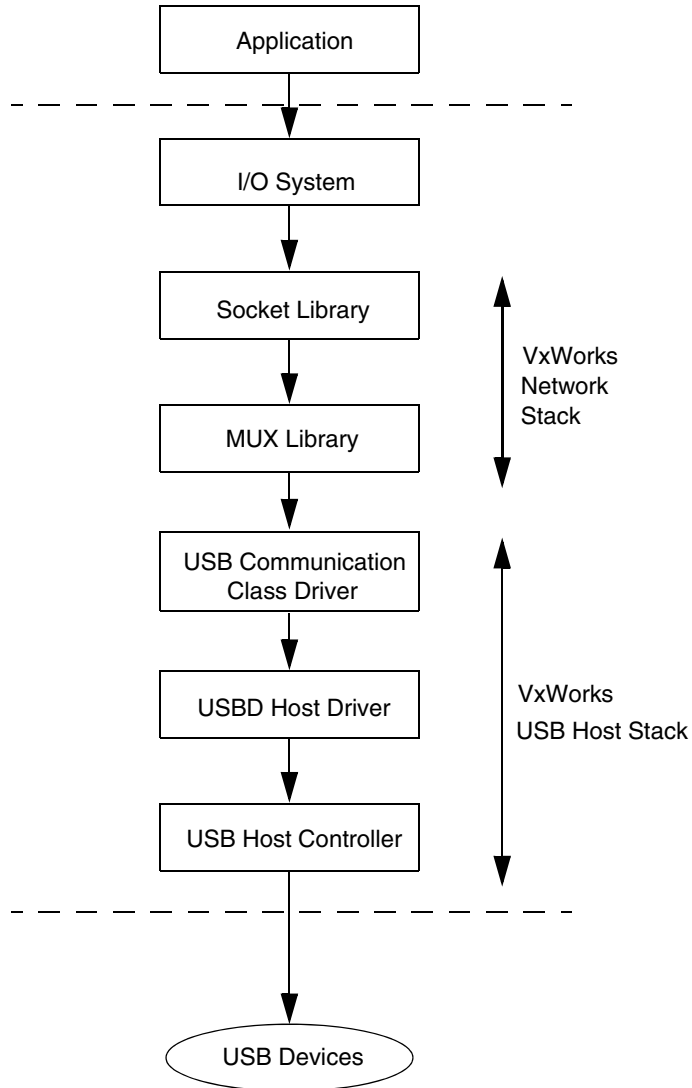
The networking control model driver presents two interfaces for transferring information to a device: a *communication class interface* and a *data class interface*.

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 cyclic redundancy check (CRC); 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.

Figure 4-6 USB Communication Class Driver Hierarchy in 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:

- Hot swap 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.

4.8.3 Dynamic Attachment

Because USB network adapters can be hot swapped 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 notifies 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, 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 USBDB and other internal resources needed for its operation. This API call also registers a callback routine with the USBDB. The callback routine is then invoked whenever a USB networking device is attached or removed.

All interactions between the USB host controller and the networking device are handled through the USBDB. Therefore, before calling **usbXXXEndInit()**, the user must ensure that the USBDB has been properly initialized with **usbdbInitialize()**. Also, before any operation with a networking device driver, the caller must ensure that at least one host controller has been attached to the USBDB through **usbUhcInit()**, **usbOhciInit()**, or **usbEhcInit()**.

4.8.5 Interrupt Behavior

The **usbXXXEndLib** driver relies on the underlying USBDB and HCD layers to communicate with USB Ethernet networking control model devices (network adapters). The USBDB 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 USBDB and HCD layers. Therefore, there is no need for the caller or BSP to connect interrupts on behalf of **usbXXXEndLib**. For the same reason, there is no post-interrupt-connect initialization code and **usbXXXEndLib** omits the **devInit2** entry point.

The **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.

4.8.6 ioctl Routines

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

4.8.7 Data Flow

The data transmission mechanism of the **usbXXXEndLib** driver deviates slightly from the **END** standard in that all data is routed through the USBBD. The **XXXEndSend()** routine fills in the **pUSB_IRP** structure and exports the structure to the USBBD 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 USBBD by the driver. When data becomes available, **XXXEndRecv()** is invoked by the IRP's callback to process the incoming packet.

Whenever data is transferred through the USBBD using the **pUSB_IRP** structure, callback routines are passed to the USBBD. 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

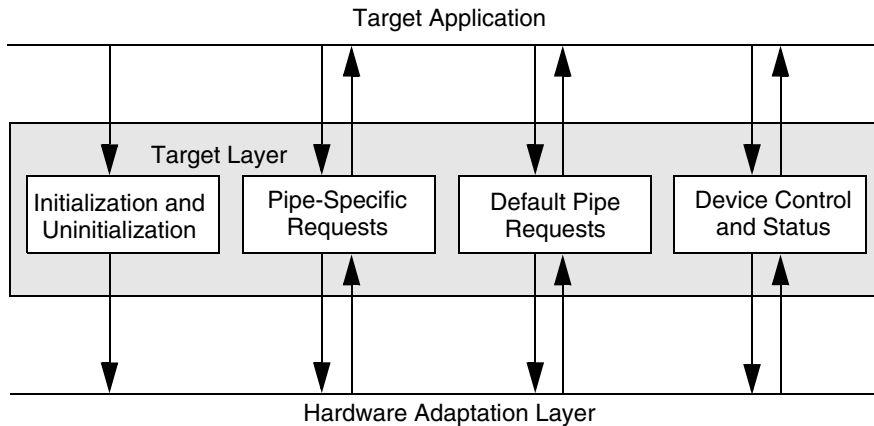
- 5.1 Introduction 97
- 5.2 Initializing the Target Layer 99
- 5.3 Attaching and Detaching a TCD 99
- 5.4 Enabling and Disabling the TCD 104
- 5.5 Implementing Target Application Callback Routines 106
- 5.6 Pipe-Specific Requests 121
- 5.7 Device Control and Status Information 131
- 5.8 Shutdown Procedure 133

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.

Figure 5-1 The Target Layer



To communicate through the target layer, the application in the configlet and your target application must coordinate to do the following:

- Initialize the target layer (see [5.2 Initializing the Target Layer](#), p.99).
- Implement certain callback routines (see [5.5 Implementing Target Application Callback Routines](#), p.106).
- Attach to a TCD (see [5.3 Attaching and Detaching a TCD](#), p.99).
- Enable the TCD (see [5.4 Enabling and Disabling the TCD](#), p.104).
- Create pipes (see [5.6.1 Creating and Destroying the Pipes](#), p.121).
- Transfer data (see [5.6.2 Transferring and Aborting Data](#), p.124).



NOTE: The target application implements all control requests, as specified in Section 9.4 *Standard Device Requests* in the USB 2.0 specification.

5.2 Initializing the Target Layer

An application in the configlette initializes the target layer in two stages as follows:

1. The configlette application 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 increments a usage count for each successful call to **usbTargInitialize()** and decrements it for each corresponding call to **usbTargShutdown()**, a routine that also requires no parameters. The target layer truly initializes itself only when the usage count goes from zero to one, and it truly shuts down only when that usage count returns to zero.
2. The configlette application attaches the target application to at least one TCD with the **usbTargTcdAttach()** routine.

Once the TCD is attached, the target application enables the TCD by calling **usbTargEnable()**.

5.3 Attaching and Detaching a TCD

Before the target application can receive and respond to requests from the host, the application in the configlette must attach the target application 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.136)
- a pointer to the TCD-defined parameters
- a pointer to the target application callback table (see [5.3.3 Target Application Callback Table](#), p.102)
- 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 attaches itself to 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 enables the TCD by calling **usbTargEnable()**. See [5.4 Enabling and Disabling the TCD](#), p.104, for the interface definition for **usbTargEnable()**.

5.3.1 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 configlet routines available in the *installDir/target/config/comps/src/usrUsbTargPciInit.c* file. The target application must call these configlet routines to populate the TCD-defined parameters. For more information, see [2.5.2 Initializing the USB Peripheral Stack Hardware](#), p.27.



NOTE: The target application must fill the TCD-defined parameters before 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()**:

```
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 );
```


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

```
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 the NET2280, it can take the form:

```
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 configlet 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.136) with the function code **TCD_FNC_ATTACH** and passes the pointer to this same set of TCD-defined parameters (see [6.5.1 Attaching the TCD](#), p.138).

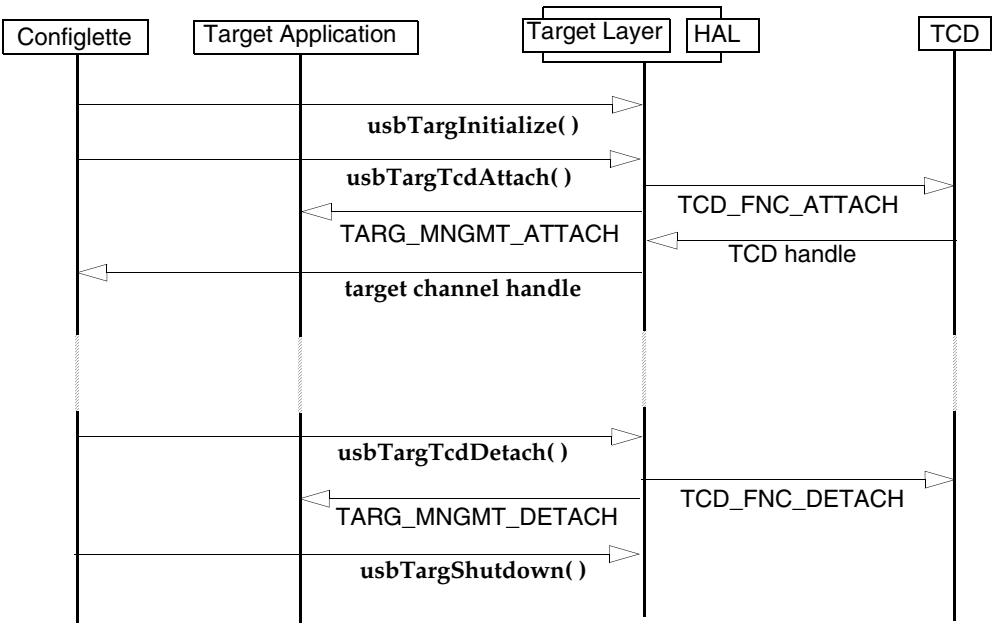
5.3.2 Detaching a TCD

The application in the configlet detaches the target application from a TCD by using **usbTargTcdDetach()**. Pass this routine handle to the target channel 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.140).

[Figure 5-2](#) illustrates how the target application attaches to and detaches from a TCD.

Figure 5-2 Attaching and Detaching a TCD



5.3.3 Target Application Callback Table

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 5-1 Target Application Callback Table

<code>mngmtFunc()</code>	<code>USB_TARG_MANAGEMENT_FUNC</code>
<code>featureClear()</code>	<code>USB_TARG_FEATURE_CLEAR_FUNC</code>
<code>featureSet()</code>	<code>USB_TARG_FEATURE_SET_FUNC</code>
<code>configurationGet()</code>	<code>USB_TARG_CONFIGURATION_GET_FUNC</code>

Table 5-1 Target Application Callback Table (cont'd)

configurationSet()	USB_TARG_CONFIGURATION_SET_FUNC
descriptorGet()	USB_TARG_DESCRIPTOR_GET_FUNC
descriptorSet()	USB_TARG_DESCRIPTOR_SET_FUNC
interfaceGet()	USB_TARG_INTERFACE_GET_FUNC
interfaceSet()	USB_TARG_INTERFACE_SET_FUNC
statusGet()	USB_TARG_STATUS_GET_FUNC
addressSet()	USB_TARG_ADDRESS_SET_FUNC
synchFrameGet()	USB_TARG_SYNCH_FRAME_GET_FUNC
vendorSpecific()	USB_TARG_VENDOR_SPECIFIC_FUNC

Before the application in the configlet calls the **usbTargTcdAttach()** routine, the target application sets 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 configlet 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 provides pointers in this table for each routine 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:

```

/*****
USB_TCD_NET2280_PARAMS      paramsNET2280;
pUSB_TARG_CALLBACK_TABLE   callbackTable;

/* Gets the Target Application Callback Table Information */

usbTargMsCallbackInfo (&callbackTable, &callbackParam);
/* Initialize usbTargLib */

```

```
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 ();
/* Populate the TCD - Defined Strucutue */

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

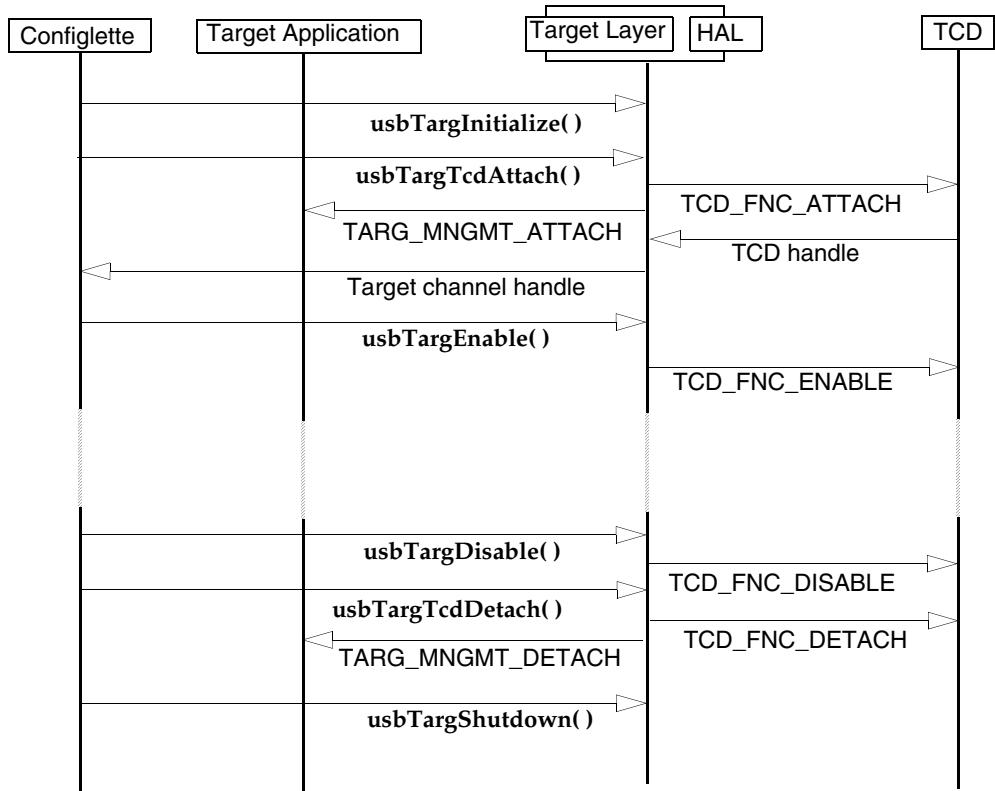
The target application may not yet be ready to begin handling USB requests when the application in the configlet calls **usbTargTcdAttach()**, so the target layer provides two routines to enable and to disable the TCD. [Figure 5-3](#) is a sequence diagram that shows how a TCD is enabled and disabled.

The application in the configlet uses **usbTargEnable()** to enable the specified TCD. The TCD, in turn, enables the underlying target controller. Until the application in the configlet calls **usbTargEnable()**, the target controller (and thus, the peripheral) are 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 configlet 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.99).

Figure 5-3 Enabling and Disabling a TCD



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.

5.5.1 Callback and Target Channel Parameters

Two parameters are common to each of the callback routines:

param

This is a parameter that is defined by the target application and that the application in the configlet passed to **usbTargTcdAttach()** as the **callbackParam**.

targChannel

This is 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**.

5.5.2 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 5-2 **Control Requests and Associated Callback Routines**

Control Request	Callback Routine	See Page . . .
CLEAR_FEATURE	featureClear()	110
SET_FEATURE	featureSet()	110
GET_CONFIGURATION	configurationGet()	113
SET_CONFIGURATION	configurationSet()	114
GET_DESCRIPTOR	descriptorGet()	114
SET_DESCRIPTOR	descriptorSet()	114

Table 5-2 Control Requests and Associated Callback Routines (cont'd)

Control Request	Callback Routine	See Page . . .
GET_INTERFACE	interfaceGet()	117
SET_INTERFACE	interfaceSet()	117
GET_STATUS	statusGet()	118
SET_ADDRESS	addressSet()	119
GET_SYNCH_FRAME	synchFrameGet()	119
<i>other</i>	endorSpecific()	120

5.5.3 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:

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

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 detachment
- 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.

```
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 *installDir/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:

```
uDeviceFeature = (USB_FEATURE_DEVICE_REMOTE | USB_FEATURE_USB20)
```



NOTE: You can also use bits three and four 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 zero to 15 corresponding to **OUT** endpoints zero to 15, and bits 15 to 31 corresponding to **IN** endpoints zero 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.108.

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 to **mngmtFunc()**. See [Context Value Parameter](#), p.108.

Depending on the operating speed of the device, the target application changes the values of various descriptors as specified in the USB 2.0 specification.

Example 5-3 Bus Reset Example

The following code fragment illustrates a bus reset event:

```
LOCAL STATUS mngmtFunc(
    pVOID param,                /* callback paramter */
    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.

5.5.4 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 returns **ERROR** under the following conditions:

- If the device is in the default state (in other words, if the device address is 0).
- If the feature does not exist or cannot be cleared (for instance, because it is the test mode feature or an unsupported feature).
- If the endpoint does not exist.

The **featureClear()** callback is defined as follows:

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

The **featureSet()** routine returns **ERROR** under the following circumstances:

- 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.
- If the endpoint or interface does not exist.

The **featureSet()** callback is defined as follows:

```
STATUS featureSet
(
    pVOID          param,
    USB_TARG_CHANNEL targChannel,
    UINT8          requestType,
    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 5-3 Standard Request Types

Request Type	Numerical Value
USB_REQ_GET_STATUS	0
USB_REQ_CLEAR_FEATURE	1
USB_REQ_GET_STATE	2
USB_REQ_SET_FEATURE	3
USB_REQ_SET_ADDRESS	5
USB_REQ_GET_DESCRIPTOR	6
USB_REQ_SET_DESCRIPTOR	7
USB_REQ_GET_CONFIGURATION	8
USB_REQ_SET_CONFIGURATION	9
USB_REQ_GET_INTERFACE	10
USB_REQ_SET_INTERFACE	11
USB_REQ_SYNCH_FRAME	12

The target application calls 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 a halt condition on the endpoint, the target application calls the **usbTargPipeStatusSet()** routine.

```
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 calls **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 calls **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 calls **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 clears or sets the feature identified by the **feature** and **index** parameters. [Table 5-4](#) lists the valid **feature** values.

Table 5-4 Feature Selectors

Feature Selector	Recipient	Feature Value
USB_FSEL_DEV_ENDPOINT_HALT	Endpoint	0
USB_FSEL_DEV_REMOTE_WAKEUP	Device	1
USB_FSEL_DEV_TEST_MODE	Device	2

Index Parameter

If the recipient of the request, as shown in [Table 5-4](#), is a device, the **index** parameter is 0; if the recipient is an endpoint, then **index** is the address of that endpoint.

5.5.5 configurationGet() Callback

The target layer calls the **configurationGet()** target application callback in response to a GET_CONFIGURATION request from the host. The **configurationGet()** callback is defined as follows:

```
STATUS configurationGet
(
    pVOID          param,
    USB_TARG_CHANNEL targChannel,
    pUINT8         pConfiguration
);
```

In the **configurationGet()** callback, the target application fills the **pConfiguration** buffer with a **UINT8** value that matches the configuration value in the current configuration descriptor.

On return from **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 0), the target application returns **ERROR** from **configurationGet()**.

5.5.6 configurationSet() Callback

The target layer calls the **configurationSet()** target application callback in response to a **SET_CONFIGURATION** request from the host. The **configurationSet()** callback is defined as follows:

```
STATUS configurationSet
(
    pVOID          param,
    USB_TARG_CHANNEL targChannel,
    UINT8          configuration
);
```

In the **configurationSet()** callback, the target application sets the current configuration to the one indicated by the **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 returns **ERROR** from **configurationSet()**.

5.5.7 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:

```
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:

```
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.

Table 5-5 USB Request Types

Request Type	Numerical Value
USB_RT_HOST_TO_DEV	0x00
USB_RT_DEV_TO_HOST	0x80
USB_RT_STANDARD	0x00
USB_RT_CLASS	0x20
USB_RT_VENDOR	0x40
USB_RT_DEVICE	0x00
USB_RT_INTERFACE	0x01
USB_RT_ENDPOINT	0x02
USB_RT_OTHER	0x03

Bit seven 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 five and six 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 zero through four 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 descriptor when multiple descriptors of the same type are implemented on a device).

Table 5-6 **Descriptor Types**

Descriptor Type	Numerical Value
USB_DESCR_DEVICE	0x01
USB_DESCR_CONFIGURATION	0x02
USB_DESCR_STRING	0x03
USB_DESCR_INTERFACE	0x04
USB_DESCR_ENDPOINT	0x05
USB_DESCR_DEVICE_QUALIFIER	0x06
USB_DESCR_OTHER_SPEED_CONFIGURATION	0x07
USB_DESCR_INTERFACE_POWER	0x08
USB_DESCR_HID	0x21
USB_DESCR_REPORT	0x22
USB_DESCR_PHYSICAL	0x23
USB_DESCR_HUB	0x29

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 fills **pBfr** with the requested descriptor (truncated if it is larger than the buffer size, which is specified by

length). The target application also sets **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 returns **ERROR** from this callback, which triggers a request error.

If the target application supports the **SET_DESCRIPTOR** request, it sets descriptors with the specified values in the **descriptorSet()** callback routine. The **length** parameter indicates the length of the descriptor that the host will send in the data stage, as specified in the setup packet.

The **descriptorSet()** callback sets **pActLen** to the actual length of the descriptor that it set. If the two values, **length** and **pActLen**, are not identical, **descriptorSet()** returns **ERROR**, which causes the target layer to stall the control endpoints.

5.5.8 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:

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

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

5.5.9 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:

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

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

5.5.10 **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:

```
STATUS statusGet
(
    pVOID          param,
    USB_TARG_CHANNEL targChannel,
    UINT16         requestType,
    UINT16         index,
    UINT16         length,
    pUINT8         pBfr
);
```

If the target application supports this callback, it stores the status of the recipient identified by **requestType** in **pBfr**. If the recipient is an endpoint, **index** specifies its address. The **length** parameter is set to two, the same value as the **wLength** value of the setup packet. For appropriate status values and their meanings, see [Table 5-7](#).

Table 5-7 **Recipient Status and Associated Status Values**

Recipient Status	Associated Value
The recipient is a device that is not self-powered and is not enabled to request remote wakeup.	0
The recipient is a device that is self-powered, but is not enabled to request remote wakeup.	USB_DEV_STS_LOCAL_POWER
The recipient is a device that is not self-powered, but is enabled to request remote wakeup.	USB_DEV_STS_REMOTE_WAKEUP

Table 5-7 Recipient Status and Associated Status Values (cont'd)

Recipient Status	Associated Value
The recipient is a device that is self-powered and enabled to request remote wakeup.	USB_DEV_STS_LOCAL_POWER USB_DEV_STS_REMOTE_WAKEUP
The recipient is an interface.	0
The recipient is an endpoint that is not currently halted.	0
The recipient is an endpoint that is currently halted.	USB_ENDPOINT_STS_HALT

The target driver sends the status back to the USB host. The target application returns **ERROR** from the **statusGet()** callback if the device is in the default state or if the **requestType** is unsupported.

5.5.11 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:

```
STATUS addressSet
(
    pVOID          param,
    USB_TARG_CHANNEL targChannel,
    UINT16         deviceAddress
);
```

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

5.5.12 synchFrameGet() Callback

It is sometimes necessary for clients to resynchronize 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:

```
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 returns **ERROR**, which triggers a request error.

5.5.13 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:

```
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 transfers, 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.130).

5.6 Pipe-Specific Requests

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

```
typedef struct targPipe          /* TARG_PIPE */
{
    USB_TARG_PIPE pipeHandle;      /* pipe handle information */
    pVOID          pHalPipeHandle; /* HAL specific pipe handle */
    pTARG_TCD      pTargTcd;       /* pointer to targ_tcd data structure*/
} TARG_PIPE, *pTARG_PIPE;
```

The members of this structure hold the following information:

pipeHandle

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

pHalPipeHandle

This is 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 the structure **USB_HAL_PIPE_INFO** is stored in this handle. This abstracts all the endpoint-specific information from 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 a 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 following:

- the endpoint descriptor (for details about this structure, see [Endpoint Descriptor](#), p.122)
- the configuration value of the device
- the number of the interface that supports the endpoint
- 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 target request block (TRB). The TRB contains the code **TCD_FNC_ENDPOINT_ASSIGN** and the endpoint descriptor, configuration value, interface number, and alternate setting. For details about this TRB, see [6.5.8 Assigning the Endpoints](#), p.142.

The default control pipes (endpoint 0) are created and maintained by the target layer 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.130).

Endpoint Descriptor

The endpoint descriptor is a **USB_ENDPOINT_DESCR** structure containing the elements the following elements:

length

This is a UINT8 containing the total length in bytes of the endpoint descriptor

descriptorType

This is a UINT8 containing the endpoint descriptor type:

USB_DESCR_ENDPOINT

endpointAddress

This is a UINT8 containing the address of the endpoint, encoded as follows:

- bits three to zero are the endpoint number
- bits six to four are reserved
- bit seven is the direction (ignored for control endpoints); zero=out, one=in

attributes

This is a UINT8 containing the endpoint attributes when it is configured using the *bConfigurationValue*:

- bits one to zero are transfer type
 - 00 – control
 - 01 – isochronous
 - 10 – bulk
 - 11 – interrupt

If this is not an isochronous endpoint, bits five to two are reserved and must be set to zero; if isochronous, they are defined as follows:

- bits three to two are synchronization type
 - 00 – no synchronization
 - 01 – asynchronous
 - 10 – adaptive
 - 11 – synchronous
- bits five to four are 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

This is 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 through its normal, non-USB-defined mechanisms.

For all endpoints, bits ten through zero specify the maximum packet size in bytes.

For highspeed isochronous and interrupt endpoints, bits twelve and eleven indicate the number of additional transaction opportunities per microframe:

- 00 – one additional (two per microframe)
- 10 – two additional (three per microframe)

- 11 – reserved
- bits 15 through 13 are reserved and must be set to 0

interval

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

- For full- and high-speed isochronous endpoints, this value must fall in the range from one to 16.
- For full- and low-speed interrupt endpoints, the value must fall in the range from one to 255.
- For high-speed interrupt endpoints, the value must fall in the range from one 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 one NAK per **interval** number of microframes. This value may range from 0 to 255.

usbTargPipeDestroy()

The target application tears down a previously-created pipe by calling **usbTargPipeDestroy()**. 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()** for generic data transfer between the USB peripheral stack and the USB host. See [Figure 5-4](#).

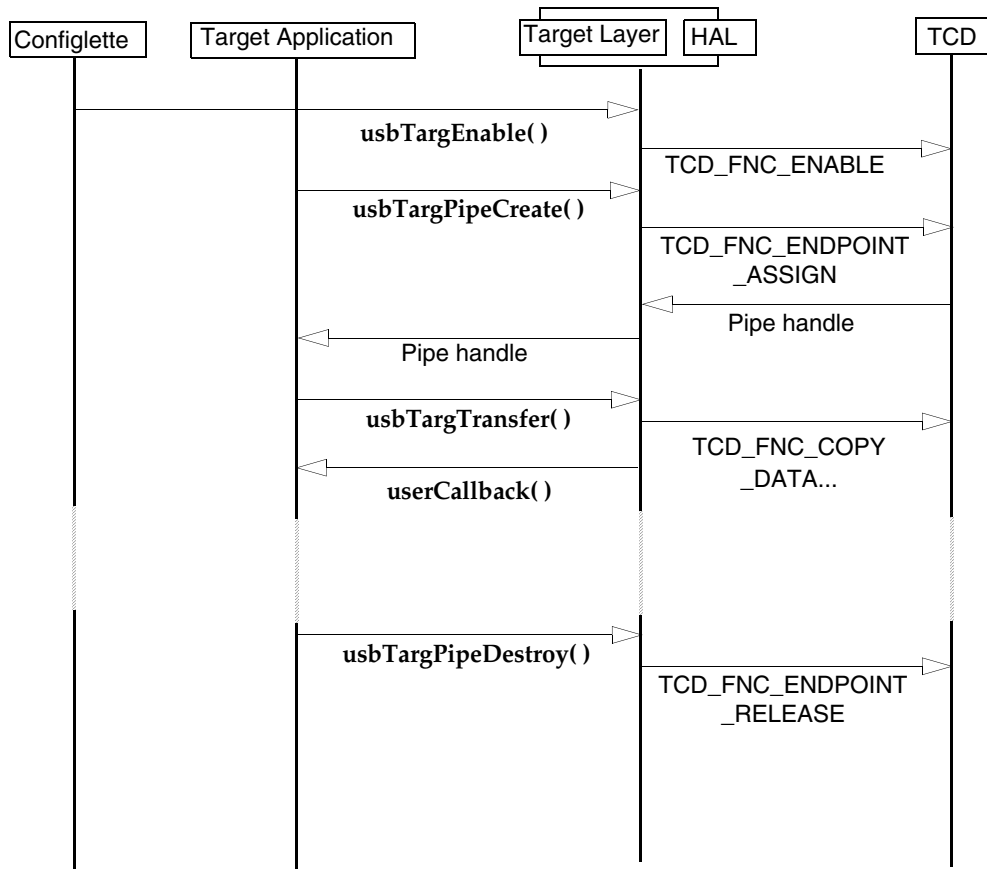
Control data transfer

If the target application instead wants to communicate with the host using the default control pipes, it does so by using the

usbTargControlResponseSend(), **usbTargControlStatusSend()**, and **usbTargControlPayloadRcv()** routines (see [5.6.4 Handling Default Pipe Requests](#), p.130).

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 **usbTargTransfer()**.

Figure 5-4 Transferring Data over a TCD



USB_ERP Structure

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

```
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 */
    int result;              /* ERP completion result: S_usbTcdLib_xxxx */
    ERP_CALLBACK targCallback; /* completion callback routine */
    ERP_CALLBACK userCallback; /* client's completion callback routine */
    pVOID pPipeHandle;       /* Pipe handle */
    UINT16 transferType;     /* type of ERP: control, bulk, etc. */
    UINT16 dataToggle;       /* ERP should start with DATA0/DATA1. */
    UINT16 bfrCount;         /* indicates count of buffers in BfrList */
    USB_BFR_LIST bfrList [1];
} USB_ERP, *pUSB_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, but 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:

```
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

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_usbTcdLib_STALL_ERROR

S_usbTcdLib_NEW_SETUP_PACKET

S_usbTcdLib_DATA_OVERRUN



NOTE: Note that the target application does 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 point 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 checks the **result** field of the ERP in this callback routine and responds 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 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

Element	Purpose
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 must have only 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	Contains the contents of the buffer.
bfrLen	Contains 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 checks this value in the userCallback() function.

usbTargTransfer() Routine

The first parameter of **usbTargTransfer()** is the handle of 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 puts the ERP in the queue of ERPs 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. The result field is set to OK for a successful transfer. The target application callback routine is 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** is called. The **result**

element of the ERP is set to `S_usbTcdLib_ERP_CANCELLED` and the `actLen` element indicates whether any of the data in the ERP was transferred.

5.6.3 Stalling and Unstalling the Endpoint

The target application may stall or un stall the generic endpoints. It stalls 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`.



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.

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 0 otherwise).

5.6.4 Handling Default Pipe Requests

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

- `usbTargControlResponseSend()`
- `usbTargControlStatusSend()`
- `usbTargControlPayloadRcv()`

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 **usbTargControlPayloadRcv()** to register a callback routine to receive control pipe responses from the host. Then, when the target application receives control data from the host, the callback routine is invoked and the **pBfr** points to the control data.

5.7 Device Control and Status Information

5.7.1 Getting the Frame Number

Some target applications, particularly those that implement isochronous data transfers, must 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 and test mode features. 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 test mode cannot be cleared.)

The second parameter of `usbTargDeviceFeatureSet()` is `ufeatureSelector`, which specifies setting the remote wakeup or 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`.

Table 5-9 **Test Selectors**

Name	Value
USB_TEST_MODE_J_STATE	1
USB_TEST_MODE_K_STATE	2
USB_TEST_MODE_SE0_ACK	3
USB_TEST_MODE_TEST_PACKET	4
USB_TEST_MODE_TEST_FORCE_ENABLE	5



NOTE: According to the USB 2.0 specification, the test mode feature cannot be cleared.

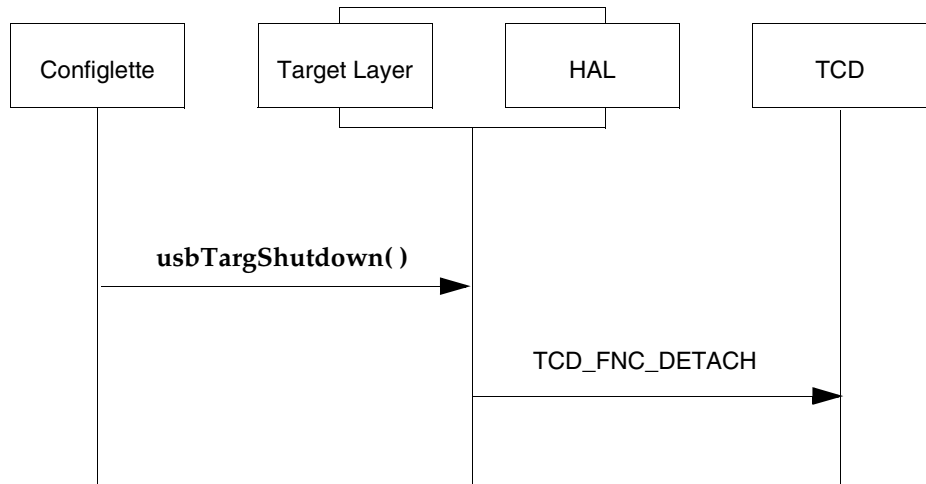
The HAL responds to this routine by calling the single entry point of the TCD with the code `TCD_FNC_DEVICE_FEATURE_CLEAR`.

5.8 Shutdown Procedure

The target layer exposes the routine **usbTargShutdown()**. This routine can be used by the target application or the configlet to shut down the USB peripheral stack.

The target layer maintains a usage count **initCount**, which is incremented every time **usbTargInitialize()** is called. Similarly, the usage count is decremented on every corresponding call to **usbTargShutdown()**. Only when the usage count reaches zero 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.

Figure 5-5 Target Layer Shutdown



6

Target Controller Drivers

- 6.1 Introduction 135
- 6.2 Hardware Adaptation Layer Overview 136
- 6.3 Single Entry Point 136
- 6.4 Target Request Block 136
- 6.5 Function Codes 138

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 (TCD) and the target layer in the USB 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 the 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 target request block (TRB). The code describes the services requested from the TCD. [6.4 Target Request Block](#), p.136, explains the TRB and [6.5 Function Codes](#), p.138, explains the 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 every 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 entry point, which has the form:

```
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 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:

```
typedef struct trb_header  
{
```

```
TCD_HANDLE handle;           /* caller's TCD client handle */
UINT16 function;            /* TCD function code */
UINT16 trbLength;          /* IN: Length of the total TRB */
} TRB_HEADER, *pTRB_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

Function Code	See Page . . .
TCD_FNC_ATTACH	138
TCD_FNC_DETACH	140
TCD_FNC_ENABLE	140
TCD_FNC_DISABLE	140
TCD_FNC_ADDRESS_SET	141
TCD_FNC_SIGNAL_RESUME	141
TCD_FNC_DEVICE_FEATURE_SET	141
TCD_FNC_DEVICE_FEATURE_CLEAR	141
TCD_FNC_CURRENT_FRAME_GET	142
TCD_FNC_ENDPOINT_ASSIGN	142
TCD_FNC_ENDPOINT_RELEASE	143
TCD_FNC_ENDPOINT_STATE_SET	143
TCD_FNC_ENDPOINT_STATUS_GET	144
TCD_FNC_IS_BUFFER_EMPTY	145
TCD_FNC_COPY_DATA_FROM_EPBUF	144
TCD_FNC_COPY_DATA_TO_EPBUF	144

Table 6-1 **TCD Function Requests** (cont'd)

Function Code	See Page . . .
TCD_FNC_INTERRUPT_STATUS_GET	145
TCD_FNC_INTERRUPT_STATUS_CLEAR	145
TCD_FNC_ENDPOINT_INTERRUPT_STATUS_GET	146
TCD_FNC_ENDPOINT_INTERRUPT_STATUS_CLEAR	148
TCD_FNC_HANDLE_DISCONNECT_INTERRUPT	148
TCD_FNC_HANDLE_RESET_INTERRUPT	148
TCD_FNC_HANDLE_RESUME_INTERRUPT	148
TCD_FNC_HANDLE_SUSPEND_INTERRUPT	148

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.

6.5 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 entry point by passing the following TRB:

```
typedef struct trb_attach
{
    TRB_HEADER header;
    pVOID tcdParam;
    USB_HAL_ISR_CALLBACK usbHalIsr;
    pVOID usbHalIsrParam;
    pUSBHAL_DEVICE_INFO pHalDeviceInfo;
```

```
pUSB_APPLN_DEVICE_INFO pDeviceInfo;
} TRB_ATTACH, *pTRB_ATTACH;
```

The **tcdParam** element points to the TCD-defined parameters structure that the application in the configlet passes to **usbTargTcdAttach()** (see [5.3.1 TCD-Defined Parameters](#), p.100). The **usbHallIsr()** is the interrupt service routine defined in the HAL. The interrupt service routine of the TCD calls this routine, passing it the **usbHallIsrParam**, whenever an interrupt event occurs.

This callback uses the following prototype:

```
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.

```
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:

```
typedef struct usb_appln_device_info
{
    UINT32 uDeviceFeature;           /* bitmap giving features supported */
                                    /* by device */
    UINT32 uEndpointNumberBitmap; /* bitmap giving endpoint numbers */
                                    /* supported by device */
} 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:

```
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 zero to 15 corresponding to OUT endpoints zero to 15, and bits 15 to 31 corresponding to IN endpoints zero to 15.

During the TCD attachment process, the TCD sets the registers of the target controller with the 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 [5.5.3 *mngmtFunc\(\)* Callback](#), p.107). 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.

```
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:

```
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:

```
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 below for this purpose.

```
typedef struct trb_signal_resume
{
    TRB_HEADER header;           /* TRB header */
} TRB_SIGNAL_RESUME, *pTRB_SIGNAL_RESUME;
```

If the USB is not 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 the TRB shown below:

```
typedef struct trb_device_feature_set_clear
{
    TRB_HEADER header;
    UINT16 uFeatureSelector;
    UINT8 uTestSelector;
```

```
} TRB_DEVICE_FEATURE_SET_CLEAR,    *pTRB_DEVICE_FEATURE_SET_CLEAR;
```

The **uFeatureSelector** element specifies the remote wakeup feature or the test mode feature (see [Table 5-4](#) for a list of feature selector values). The **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 resynchronize 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 **frameNo**). 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 **TCD_FNC_CURRENT_FRAME_GET** code is shown below:

```
typedef struct trb_current_frame_get
{
    TRB_HEADER header;
    UINT16 frameNo;
} TRB_CURRENT_FRAME_GET, *pTRB_CURRENT_FRAME_GET;
```

In response to the **TCD_FNC_CURRENT_FRAME_GET** request, the TCD returns the current USB frame number in the **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. 122). The HAL maintains all the endpoint-specific information in the **USBHAL_PIPE_INFO** structure, shown below:

```
typedef struct usbhal_pipe_info
{
    UINT8 uEndpointAddress;
    UINT8 uTransferType;
    UINT32 pipeHandle;
    LIST_HEAD listHead;
    MUTEX_HANDLE mutexHandle;
} USBHAL_PIPE_INFO, *pUSBHAL_PIPE_INFO;
```



NOTE: The creation of generic pipes depends entirely on the target application. Generally, the target application creates the generic endpoints on receiving the **SET_CONFIGURATION** request from the host.

The HAL calls the single entry point of the TCD with the TRB shown below:

```
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;
```

The HAL also calls it with the function code **TCD_FNC_ENDPOINT_ASSIGN** (see [5.6.1 Creating and Destroying the Pipes](#), p.121). 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.122.

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:

```
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 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:

```
typedef struct trb_endpoint_state_set
{
    TRB_HEADER header;
    UINT32 pipeHandle;
    UINT16 state;
} TRB_ENDPOINT_STATE_SET, *PTRB_ENDPOINT_STATE_SET;
```

The state element is either **TCD_ENDPOINT_STALL** or **TCD_ENDPOINT_UNSTALL**.

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:

```
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 Cancelling ERPs

Once the pipes are created, they may be used to transfer data (see [5.6.2 Transferring and Aborting Data](#), p.124). 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:

```
typedef struct trb_copy_data_from_epbuf
{
    TRB_HEADER header;
    UINT32 pipeHandle;
    pUCHAR pBuffer;
    UINT32 uActLength;
} TRB_COPY_DATA_FROM_EPBUF, *PTRB_COPY_DATA_FROM_EPBUF;

typedef struct trb_copy_data_to_epbuf
{

```

```
TRB_HEADER header;
UINT32 pipeHandle;
pUCHAR 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 from that indicated by **uActLength**, the TCD overwrites **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. 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 code **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;
```

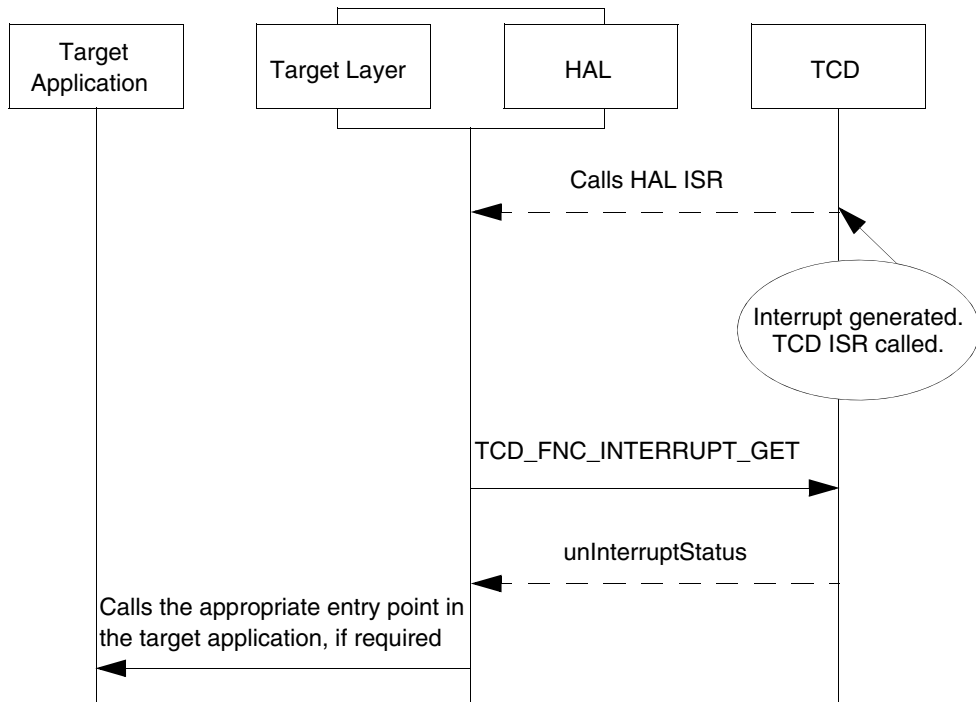
In response to the **TCD_FNC_INTERRUPT_STATUS_GET**, the TCD indicates the types of interrupts that are pending by setting the **uInterruptStatus** element of the TRB. This element is a bitmap with the following form:

- 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:

Figure 6-1 Handling Interrupts



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 what type of interrupt occurred. The TRB for this process is shown below:

```
typedef struct trb_endpoint_interrupt_status_get
{
    TRB_HEADER header;
    UINT32 pipeHandle;
    UINT32 uEndptInterruptStatus;
} TRB_ENDPOINT_INTERRUPT_STATUS_GET, *pTRB_ENDPOINT_INTERRUPT_STATUS_GET;
```

uEndptInterruptStatus is a bitmap that indicates the types of interrupt that have occurred on that endpoint. It takes the following form:

- bit zero – endpoint Interrupt
- bit one – setup Interrupt
- bit two – OUT interrupt
- bit three – IN interrupt

The TCD uses bits four to eleven to report errors. These errors are defined in **usbTcd.h** and are shown in [Table 6-2](#):

Table 6-2 Endpoint Interrupt Status Errors

Error	Value
USBTCD_ENDPOINT_TRANSFER_SUCCESS	0x0000
USBTCD_ENDPOINT_DATA_TOGGLE_ERROR	0x00E0
USBTCD_ENDPOINT_PID_MISMATCH	0x00F0
USBTCD_ENDPOINT_COMMUN_FAULT	0x0100
USBTCD_ENDPOINT_STALL_ERROR	0x0110
USBTCD_ENDPOINT_DATA_OVERRUN	0x0120
USBTCD_ENDPOINT_DATA_UNDERRUN	0x0130
USBTCD_ENDPOINT_CRC_ERROR	0x0140
USBTCD_ENDPOINT_TIMEOUT_ERROR	0x0150
USBTCD_ENDPOINT_BIT_STUFF_ERROR	0x0160

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:

```
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 events 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 uses these notices only 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:

```
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:

```
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

BSP Porting

7.1 Configuring the USB Peripheral Stack

The USB peripheral stack takes advantage of the non-VxBus PCI configuration capabilities in VxBus. To access these functions the macro `INCLUDE_PCI_OLD_CONFIG_ROUTINES` must be defined, typically in `config.h` in the BSP.

7.1.1 Initialization of the USB Peripheral Stack

USB peripheral stack initialization starts with the initialization of the target controllers. The routine `usrUsbTargXXXInit()`, defined in *installDir/vxWorks-6.x/target/config/comps/src*, calls the USB target hardware configlet routine of the corresponding target controller, which is defined in *installDir/vxworks6.x/target/config/comps/src/usrUsbTargPciInit.c*.

The following are the hardware initialization routines:

- `sysIsp1582PciInit()` – This configures the PCI-based Philips ISP1582 target controller
- `sys2NET2280PciInit()` – This configures the PCI-based Netchip NET2280 target controller

Configuring the peripheral hardware involves the following:

- Determining the base address map of the controller.
- Determining the interrupt request number of the controller.

Example Initializing Resources for a NET2280 controller

The following sample code demonstrates how to initialize the resources for the PCI- based NET2280 target controller:

```
/*
 * 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;
}
```

7.1.2 Creating a BSP-Specific Stub File for the USB Peripheral Stack

The BSP-specific stub files provide the 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 peripherals.

Eight-, 16- and 32-Bit Data I/O

The target controller driver (TCD) uses the routines listed below for read and write operations with the I/O-mapped PCI-based target controller.

The following routines provide data lengths such eight-, 16- and 32-bit, for accessing the target controller registers of varying data widths.

- **usbPciByteIn ()** – Reads a byte from PCI I/O space.

- **usbPciWordIn()** – Reads a word from PCI I/O space.
- **usbPciDwordIn()** – Reads a dword from PCI I/O space.
- **usbPciByteOut()** – Writes a byte to PCI I/O space.
- **usbPciWordOut()** – Writes a word to PCI I/O space.
- **usbPciDwordOut()** – Writes a dword to PCI I/O space.

Interrupt Routines

The following routines are used to connect and disconnect the ISR with the corresponding interrupts.

- **usbPciIntConnect()** – Connects to the interrupt line.
- **usbPciIntRestore()** – Disconnects from the interrupt line.

9

usbTool Code Exerciser Utility Tool

- 9.1 Introduction 155
- 9.2 Running usbTool from the Shell 156
- 9.3 Using the usbTool Execution Sequence 156
- 9.4 Testing Applications 157

9.1 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.



NOTE: The **usbTool** utility relies internally on static data. Do not run multiple instances of **usbTool** simultaneously.

9.2 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.

9.3 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 that produced by 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
```




NOTE: Selecting any of the device driver components (**keyboard**, **mouse**, **printer**, or **speaker**) includes the device test commands into **usbTool**. If the components are not included, the commands are not available.

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

spkrInit, **spkrDown**, **spkrFmt**, **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**.

9.4 Testing Applications

This section describes how to test applications by using the **usbTool** utility.

9.4.1 Testing the Keyboard Application

To test the keyboard application, proceed as follows:

Step 1: Start the keyboard test application.

Invoke the **usbTool** on the target.

Enter the **targInit TCD kbd** command at the prompt, where TCD can be **d21**, **isp1582**, or **net22280**. The following example shows this command and the resulting output for a PDIUSB12 controller:

```
usb> targInit d12 kbd
usbTargInitialize() returned OK.
Philips PDIUSB12: ioBase = 0x365, irq = 7, dma = 3
usbTargTcdAttach() returned OK
targChannel 0x2342
usbTargEnable() returned OK.
```

Step 2: Connect the target with the host and invoke usbTool.

Connect the target containing the USB peripheral stack that has the keyboard functionality to the USB host stack with a USB cable.

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 xhci** (where *x* is "o", "u", or "e").

Step 4: Issue the usbEnum command.

Enter the **usbEnum** command at the **usbTool** prompt on the host. 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 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 entering the **kdbInit** command at the **usbTool** prompt.

Step 6: Poll the driver.

Enter the **kdbPoll** command from the **usbTool** prompt at the host to poll the keyboard SIO driver for input.

Give the **kbdReport** command from the target shell. 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
```

9.4.2 Testing the Printer Application

Test the printer application as follows:

Step 1: Start the printer test application.

Invoke the **usbTool** on the target.

Enter the **targInit TCD prn** command from its prompt, where TCD can be **d21**, **isp1582**, or **net22280**. The following example shows this command and the resulting output for a PDIUSB12 controller:

```
usb> targInit d12 prn
usbTargInitialize() returned OK
Philips PDIUSB12: ioBase = 0x368, irq = 7, dma = 3
usbTargTcdAttach() returned OK
targChannel = 0x1
```

```
usbTargEnable() returned OK
```

Step 2: Connect the target with the host.

Connect the target containing the USB peripheral stack that has the printer functionality to the USB host stack with a USB cable.

Step 3: Initialize the host and attach the controller.

Initialize the USB host stack with the **usb**i command.

Attach the host controller with the command **attach xhci** (where *x* is "o", "u", or "e").

Step 4: Issue the usbEnum command.

Enter the **usbEnum** command at the **usbTool** prompt on the host. 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 3not connected
      port 3 not connected
      port 4 not connected
```

Step 5: Initialize the printer driver.

Initialize the printer driver on the host by entering the **prnInit** command at the **usbTool** prompt.

Print 4,096-byte blocks of data by giving the following command from the host (where *n* can be 1, 2, 3, and so on):

```
usb> print4k n
```

Step 6: Issue the prnDump command.

Give the **prnDump** command from the target peripheral. This results in output on the target like the following:

```
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.[.]_
00 61 00 63 00 65 00 67 00 69 00 6b 00 6d 00 6f .a.c.e.g.i.k.m.o
00 71 00 73 00 75 00 77 00 79 00 7b 00 7d 00 7f .q.s.u.w.y.{}..
```

and in output on the host like that 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” appears instead.

9.4.3 Testing the Mass Storage Application

After initializing the peripheral mass storage function, the host should recognize that a mass storage device is attached, if the host has mass storage class driver initialization built in. You can then do normal file system operations on the USB disk, as in the following:

```
cd("/bd0")
creat("temp.txt",2)
rm("temp.txt")
```


A

Glossary

[A.1 Glossary Terms 163](#)

[A.2 Abbreviations and Acronyms 176](#)

A.1 Glossary Terms

active device

An *active device* is a device that is powered and that is not in the *suspend* state.

attach

To *attach* the target application and the TCD so that the target application can receive and respond to requests from the host through that TCD, the application in the configlet 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 *installDir/target/config/BSP/usbPciStub.c*.

build flag

A *build flag* variable is defined in the makefile and used to set various compiler options.

build macro

A *build macro* is a [build flag](#).

bulk transfer

Bulk transfer is one of the four USB transfers. These are nonperiodic, large bursty communications typically used for a transfer that can use any available bandwidth but 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](#).

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 provider and consumer of the transferred data.

compile-time macro

A *compile-time macro* is a [build flag](#).

configlet

A *configlet* is a VxWorks configuration routine that initializes the USB components. All configlet routines are located in the directory `installDir/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 bidirectional 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, such as a data/fax modem device. The term “device” is sometimes interchanged with “[peripheral](#)”.

device address

A *device address* is a value between zero 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 device capable of high speed 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 USB 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. 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 with a hardware device, that is, a device driver.

Enhanced Host Controller Interface

The *Enhanced Host Controller Interface* (EHCI) is the host controller compliant with the USB 2.0 specification.

enable

To *enable* a USB device is to bring it up in such a way that it becomes 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.122).

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 *installDir/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.112.

first in, first out

A *first in, first out* (FIFO) protocol is an approach to handling program work requests from a queue or stack. The request that enters the queue first is addressed first.

frame

A *frame* is a one-millisecond time base, established on full- and low-speed buses.

frame number

A *frame number* is an identifier for the active frame.

full-speed

Full-speed refers to USB operation at 12 Mb/s. See also [low-speed](#) and [high-speed](#).

function code

A *function code* describes the services requested from the TCD

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](#)).

halt

See [stall](#).

handle

A *handle* is a unique identifier used for communication with the object to which it is assigned. For example, the target controller driver (TCD) provides the target

application with a handle during the attachment process. The target application uses this handle to communicate with the attached TCD.

There is also a handle for 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.

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 a service provider interface (SPI) for interaction with the host controller. The host controller driver hides the specifics of the host controller hardware implementation from the higher 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 in an appropriate direction between itself (on the host) and an endpoint of a device.

IN endpoint

The *IN endpoint* is the [endpoint](#) that corresponds to IN requests from the 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 service routine

An *interrupt service routine* is a software routine that is executed in response to an interrupt.

interrupt transfer

An *interrupt transfer* is one of the four USB transfer types. 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 bidirectional 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 with the USB 1.1 specification.

OUT endpoint

The *OUT endpoint* is the endpoint that corresponds to **OUT** requests from the host.

peripheral

See [device](#).

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 USB 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 set of rules, procedures, or conventions relating to the 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](#).

reset

See [bus reset](#).

resume

A *resume* is a signal sent by the host to make a device in a suspended state come out of that state and restart. See also, [remote wakeup](#).

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 sent by 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 indicates 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 support. Devices 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 microframe. An SOF allows endpoints to identify the start of the microframe and synchronize internal endpoint clocks with 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* a device is to put it into an inactive state to conserve power. For example, the host may suspend a device for a specific period when it observes that there is no traffic on the bus. While suspended, the USB device maintains its 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](#).

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 the target controller and carries out all of the hardware-specific implementation of the target controller commands.

target layer

The *target layer* is a consistent, abstract mediator between a variety of target applications and the HAL.

target request block

The *target request block* (TRB) is a request block created by the HAL. It consists of the handle of the TCD, a *function code*, the length of the TRB, and the parameter associated with the function code. Every function code has 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, an optional data packet, and an optional handshake packet. Specific packets are allowed or 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 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.29.

Universal Host Controller Interface

The *Universal Host Controller Interface* (UHCI) is the host controller compliant with the 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 controllers*. It is a hardware-independent software layer that implements 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.29.

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 causing the device to come out of a suspended state.

A.2 Abbreviations and Acronyms

The Wind River USB documentation uses the following abbreviations and acronyms:

Table A-1 **Abbreviations and Acronyms**

Acronym	Description
BSP	Board Support Package
CBI	Command-Bulk-Interrupt
DMA	Direct Memory Access
EHCI	Enhanced Host Controller Interface
END	Enhanced Network Driver
ERF	Event Reporting Framework
ERP	Endpoint Request Packet
FIFO	First In, First Out
HAL	Hardware Adaptation Layer
HC	Host Controller
HCD	Host Controller Driver
HID	Human Interface Device
HRFS	Highly Reliable File System
IDE	Integrated Development Environment
IRP	I/O Request Packet
IRQ	Interrupt Request
MPEG	Moving Pictures Experts Group
OHCI	Open Host Controller Interface
PCI	Peripheral Control Interface
SIO	Serial I/O

Table A-1 **Abbreviations and Acronyms** (cont'd)

Acronym	Description
SOF	Start-of-Frame
SPC-2	SCSI Primary Commands-2
TC	Target Controller
TCD	Target Controller Driver
TRB	Target Request Block
UHCI	Universal Host Controller Interface
UML	Unified Modeling Language
USB	Universal Serial Bus
USB D	Universal Serial Bus Driver
XBD	Extended Block Device

Index

Symbols

.wav file 19

A

abbreviations, listed 176

acronyms, listed 176

active device

glossary definition 163

actLen

USB_BFR_LIST 129

USB_ERP 130

ADDED_CFLAGS+=NET2280_DMA_
SUPPORTED 18

addressed state

interfaceGet() 117

interfaceSet() 118

addressSet() 103

callback response to SET_ADDRESS 119

table of associated control requests 107

alternateSetting

interfaceSet() 118

API routines 90

attach

glossary definition 163

attach ehci 156

attach ohci 156

attach uhci 156

attach xhci

keyboard testing 158

printer testing 160

ATTACH_USB_KEYBOARD_TO_SHELL 16

attaching a TCD

usbTargetEnable() 100

attachment

TCD to mass storage TA 103

attributes

USB_ENDPOINT_DESCR 123

audio device

glossary definition 163

audio devices 80

audio driver 26, 80

B

bandwidth

glossary definition 163

bConfigurationValue 123

bfrCount

USB_ERP 128

bfrLen

USB_BFR_LIST 129

bfrList 126, 129

bfrCount 128

erpLen 126

- pid element 129
- USB_ERP 128
- big-endian
 - glossary definition 164
- BIO structure 88
- Block_IO structure 88
- bmRequestType 120
- board support package, *see* BSP
- board support packages 6
- bRequest 120
- BSP
 - glossary definition 164
- BSPs 6
- build flag
 - glossary definition 164
- build macro
 - glossary definition 164
- bulk only
 - configuration parameters 14
- bulk transfer 2
 - glossary definition 164
- BULK_DRIVE_NAME 14
- BULK_MAX_DEVS 14
- BULK_MAX_DRV_NAME_SZ 14
- BULK_RESET_NOT_SUPPORTED 18
- bulk-only class driver 13
- Bulk-Only driver 86
- bus deregistration 59
- bus enumeration routines 48
- bus registration 59
- bus reset
 - glossary definition 164
 - TARG_MNGMT_BUS_RESET management
 - event code 107
- bus tasks, overview 36

C

- callback routine
 - glossary definition 164
- callback table, target application 102
- callbackInstall() 76
- callbackParam
 - usbTargTcdAttach() 99, 106

- callbacks 38, 43
- cancelURB() 57
- CBI
 - configuration parameters 15
- CBI driver 86
- CBI_DRIVE_NAME 15
- channel
 - glossary definition 164
- class drivers
 - list 4
- Class-specific devices 38
- CLEAR_FEATURE 110
 - clearing a device feature
 - hardware layer 141
 - target layer 132
 - featureClear() callback 110
 - request 60
 - setting and clearing device-specific features 141
 - table of control requests 106
 - usbHstClearFeature() 39
 - usbTargDeviceFeatureClear() in response to 132
- client
 - glossary definition 165
- client callback tasks 43
- client modules 36
 - registration 43
- cmdParser 32
- code exerciser
 - usbTool 155
- communication class device initialization 26
- communication class drivers 92
 - data flow 96
 - dynamic attachment 94
 - Ethernet networking control model driver 92
 - initialization 95
 - interrupt behavior 95
 - ioctl routines 96
- compile-time macro
 - glossary definition 165
- component dependencies 18
- configlette
 - glossary definition 165
- configuration parameters

- bulk only 14
- CBI 15
- keyboard 16
- pegasus 15
- configuration stage
 - addressSet() 119
- configurationGet() 102, 113
 - defined 113
 - table of associated control requests 106
- configurationSet() 103
 - defined 114
 - table of associated control requests 106
- connecting a device to a hub 64
- control data transfer 124
- control endpoint
 - glossary definition 165
- control pipe
 - glossary definition 165
- control transfer 2
 - glossary definition 165
- copy() 26
- createPipe() 56

D

- data flow 51
 - mass storage devices 91
 - microphones 86
 - mouse driver 76
 - printers 80
 - speakers 85
- data structures
 - USBHST_URB 40
- data transfer
 - pipe 53
- Data Transfer Interfaces 40, 41
- data transfer rates 2
- dataToggle
 - USB_ERP 128
- default address
 - glossary definition 165
- default control pipes 6, 122
 - glossary definition 165
- default state

- interfaceGet() 117
- interfaceSet() 118
- statusGet() 119
- definitions of terms 163
- deletePipe() 56
- demo applications
 - usbAudio 26
- descriptor
 - glossary definition 166
- descriptor type
 - table of macros 116
- descriptorGet() 103, 116
 - defined 114
 - table of associated control requests 106
- descriptorIndex
 - descriptorGet() 114
 - descriptorSet() 114
- descriptorSet() 103, 114, 117
 - defined 114
 - table of associated control requests 106
- descriptorType 116
 - descriptorGet() 114
 - descriptorSet() 114
 - parameter described 116
 - USB_ENDPOINT_DESCR 122
- detach
 - glossary definition 166
- detaching
 - TCD 101
- device
 - glossary definition 166
- device address
 - glossary definition 166
- device configuration 48
- device initialization 24
- device qualifier
 - glossary definition 166
- DeviceAdd_Callback 38
- deviceAddress
 - addressSet() 119
 - TCD_FNC_ADDRESS_SET 141
 - TRB_ADDRESS_SET 141
- DeviceRemove_Callback 38
- DeviceResume_Callback 38
- DeviceSuspend_Callback 38

- disable
 - glossary definition 166
- disabling
 - TCD 104
- disconnect
 - glossary definition 166
- disconnect event 148
- disconnect signal 107
- dma 101
- dma element 101
- DMA transfer 18
- documentation
 - tools 7
 - VxWorks 7
- downstream
 - glossary definition 166
- driver
 - glossary definition 167
- dynamic attachment registration 44
- dynamic device attachment 68
 - mass storage devices 90
 - speakers 84

E

- EHCI
 - host controller initialization 16
 - overview of host controllers 31
 - specification 3, 7
- EHCI host controller driver 12
- enable
 - glossary definition 167
- enabling
 - TCD 104
- enabling and disabling the TCD 104
- END driver
 - USB Pegasus 19
- endpoint attributes 123
- endpoint descriptor
 - described 122
 - glossary definition 167
 - setting the 122
- USB_ENDPOINT_DESCR structure 122
- endpoint interrupt

- clearing all 148
- endpoint interrupt status errors 147
- endpoint number
 - glossary definition 167
- endpoint request packet, *see* ERP
- endpointAddress
 - USB_ENDPOINT_DESCR 122
- endpointId
 - USB_ERP 128
- endpoints
 - creation 142
 - glossary definition 167
 - number supported 140
- Enhanced Host Controller Interface, *see* EHCI
- ENOSYS 71, 76
- enumerate
 - glossary definition 167
- ERF
 - device driver model
- ERP
 - completion result 127
 - type 128
- ERP_CALLBACK 128
 - userCallback() 128
- erpLen
 - USB_ERP 126
- error reporting 59
- error reporting conventions for HCD 59
- errors
 - split errors, handling 58, 59
- Ethernet networking control model driver 92
- Event Reporting Framework, *see* ERF
- Extended Block Device, *see* XBD

F

- feature
 - featureClear() 112
 - featureSet() 112
 - glossary definition 168
- feature selectors 113
- featureClear() 102, 110
 - table of associated control requests 106
- featureSet() 102, 110, 111

- table of associated control requests 106
- FEI Ethernet interface 19
- FIFO
- file system components 18
- file system support 18
- Firmware Programming Guide for NET2280 PCI
 - USB High Speed Peripheral Controller 8
- Firmware Programming Guide for PDIUSB12 8
- first in, first out, *see* FIFO
- for 3
- frame
 - glossary definition 168
- frame number
 - glossary definition 168
 - retrieving 131
- frameNo
 - TRB_CURRENT_FRAME_GET 142
- frameNo (TRB element) 142
- full data transfer rate 2
- full-speed
 - glossary definition 168
- function
 - TRB header 137
- function codes
 - glossary definition 168
 - TRBs and TCDs 138
- function driver
 - glossary definition 168

G

- generic data transfer 124
- generic endpoints
 - glossary definition 168
- GET_CONFIGURATION 39
 - request, response to 113
 - table of control requests 106
- GET_DESCRIPTOR 39, 60
 - descriptorGet() 114
 - response to a request 131
 - table of control requests 106
- GET_INTERFACE
 - control request for interfaceGet() 107
 - interfaceGet() callback 117

- usbHstGetInterface() 39
- GET_STATUS
 - getting the endpoint status 144
 - statusGet() callback 118
 - table of control requests 107
 - usbHstGetStatus() 40
- GET_SYNCH_FRAME
 - table of control requests 107
- getFrameNumber() 57

H

- HAL 5, 136
 - glossary definition 169
- halt
 - glossary definition 168
- handle
 - glossary definition 168
 - TRB header 137, 140
- hardware adaptation layer, *see* HAL
- hardwareInterfaceInit() 35
- HCD Error Reporting Conventions 59
- headset audio demonstration configlet 14
- HID_MSE_BOOT_REPORT 76
- high data transfer rate 2
- high-speed
 - glossary definition 169
- host
 - glossary definition 169
- host component parameters 12
- host controller
 - glossary definition 169
 - initialization 16, 23
 - overview of specifications 3
 - specification documentation 7
- host controller driver 34
 - deregistration 59
- EHCI 31
 - error reporting conventions 59
 - glossary definition 169
 - interface and requirements 54
- interfaces 56
- OHCI 31
 - overview 54

- registering 54
- registering a bus for the host controller UHCI 31
- host controllers
 - bus deregistration 59
 - bus registration 59
- host driver 12
- host driver component 12
- host stack
 - glossary definition 169
 - initialization 23
- hub
 - glossary definition 169
- hub class drivers 33, 62
 - registering 62
- hub functionality 31

I

I/O request packet, *see* IRP

IEEE-1284 80

IN endpoint

- glossary definition 170

INCLUDE_ATA 19

INCLUDE_AUDIO_DEMO 27

INCLUDE_DEVICE_MANAGER 18

INCLUDE_DOSFS 18

INCLUDE_DOSFS_CHKDSK 18

INCLUDE_DOSFS_FMT 18

INCLUDE_DOSFS_MAIN 18

INCLUDE_EHCI 12, 16

INCLUDE_EHCI_BUS 12, 17

INCLUDE_EHCI_INIT 12, 16

INCLUDE_ERF 18

INCLUDE_FS_MONITOR 18

INCLUDE_IFCONFIG 19

INCLUDE_KBD_EMULATOR 20, 21

INCLUDE_KBD_EMULATOR_INIT 20, 21

INCLUDE_MS_EMULATOR 20, 21

INCLUDE_MS_EMULATOR_INIT 20, 21

INCLUDE_NET2280 20, 21

INCLUDE_OHCI 12, 16, 21

INCLUDE_OHCI_BUS 12, 17

INCLUDE_OHCI_INIT 12, 16

INCLUDE_PDIUSB12 20, 21

INCLUDE_PHILIPS1582 20, 21

INCLUDE_PRN_EMULATOR 21

INCLUDE_PRN_EMULATOR_INIT 21

INCLUDE_SELECT 17

INCLUDE_UHCI 12, 16

INCLUDE_UHCI_BUS 13, 17

INCLUDE_UHCI_INIT 13, 17

INCLUDE_USB 12

- defined 12
- required 16

INCLUDE_USB_AUDIO_DEMO 14, 19

INCLUDE_USB_HEADSET_DEMO 19

INCLUDE_USB_INIT 12, 16

INCLUDE_USB_KEYBOARD 13, 17

INCLUDE_USB_KEYBOARD_INIT 13, 17

- configuration parameters 16

INCLUDE_USB_MOUSE 13, 17

INCLUDE_USB_MOUSE_INIT 13, 17

INCLUDE_USB_MS_BULKONLY 13, 17

INCLUDE_USB_MS_BULKONLY_INIT 13, 14, 17

INCLUDE_USB_MS_CBI 13, 17

- configuration parameters 15

INCLUDE_USB_MS_CBI_INIT 13, 17

INCLUDE_USB_PEGASUS_END 13, 17

INCLUDE_USB_PEGASUS_END_INIT 14, 17

- configuration parameters 15

INCLUDE_USB_PRINTER 13, 17

INCLUDE_USB_PRINTER_INIT 13, 17

INCLUDE_USB_SPEAKER 13, 17, 19

INCLUDE_USB_SPEAKER_INIT 13, 17, 19, 27

INCLUDE_USB_TARG 20, 21

INCLUDE_USBT00L 19, 21, 22

INCLUDE_XBD 18

INCLUDE_XBD_PART_LIB 18

INCLUDE_XXX_INIT 22

including the host stack component 16

index 118

- featureClear() 113
- featureSet() 113
- statusGet() 118
- vendorSpecific() 120

initCount 133

initialization

- host controller 16, 23
- host stack 23
- mass storage devices 91
- USB devices 17
- USB host stack 16
- initializing USB devices 17
- installDir 2
- interface
 - interfaceGet() 117
 - interfaceSet() 118
- interfaceGet() 103, 117
 - described 117
 - table of associated control requests 107
- interfaceSet() 103, 118
 - described 117
 - table of associated control requests 107
- interrupt behavior 95
- interrupt endpoint
 - glossary definition 170
- interrupt handling
 - diagram 146
 - endpoint-specific 146
- interrupt mode 19
- interrupt request
 - glossary definition 170
- interrupt request line
 - glossary definition 170
- interrupt request, *see* IRQ
- interrupt service routine
 - glossary definition 170
- interrupt status 145
- interrupt transfer 2
 - glossary definition 170
- interval
 - USB_ENDPOINT_DESCR 124
- ioBase 101
- ioctl routines
 - mouse driver 76
 - printer driver 80
- ioctl() 85, 86
- IP_MAX_UNITS 19
- IRP
 - glossary definition 169
- IRQ 101
 - glossary definition 170

- irq element 101
- isBandwidthAvailable() 56
- isochronous transfer 2
 - glossary definition 170
- ISP 1582 driver 18
- isRequestPending() 57

K

- kbdDown 157
- kbdInit 157
- kbdPoll 157
 - command to pool the keyboard 158
 - example 159
- kbdReport 159
- keyboard
 - application testing 157
 - configuration parameters 16
 - drivers 65
 - initialization 25
- keyboard class drivers 13, 66
 - initializing 66
 - registration 67
- keyboard drivers 65
 - data flow 71
 - initialization 66
 - ioctl routines 71
 - typematic repeat 72
- keyboard emulator 20

L

- languageId
 - descriptorGet() 114, 116
 - descriptorSet() 114, 116
- length 118
 - descriptorGet() 117
 - descriptorSet() 117
 - statusGet() 118
 - USB_ENDPOINT_DESCR 122
 - vendorSpecific() 120
- length parameter 117

LINK 126
little endian
 glossary definition 170
low data transfer rate 2
low-speed
 glossary definition 170

M

management event codes
 mngmtFunc(), parameter to 107
 TARG_MNGMT_ATTACH 108
 TARG_MNGMT_BUS_RESET 108
 TARG_MNGMT_DISCONNECT 107
 TARG_MNGMT_RESUME 107
 TARG_MNGMT_SUSPEND 108
mass storage application
 testing 161
mass storage bulk-only class driver 13
mass storage CBI class driver 13
mass storage class drivers 86
 data flow 91
 dynamic attachment 90
 initialization 91
 XBD driver model 88
mass storage devices
 data flow 91
 dynamic attachment 90
 initialization 25, 91
mass storage emulator 20, 21
mass storage function driver 18
maximum packet size 123
Maximum UFI Devices 15
maxPacketSize
 USB_ENDPOINT_DESCR 123
message pipe
 glossary definition 171
microframe
 glossary definition 171
microphones 80
 data flow 86
 recognizing and handling 85
MIDI 81
mngmtCode parameter

 mngmtFunc(), to 107
mngmtFunc() 102
mngmtFunc() callback function 107
modifyDefaultPipe() 56
mouse
 driver 73
 initialization 25
mouse class driver 13
 uninitialization 76
mouse driver
 data flow 76
 dynamic attachment 75
 initialization 73
 ioctl routines 76
mouseTest 157
MPEG 81
M-Systems FlashOnKey 26

N

NET 2280 driver 18
NET2280
 configlet routine 27
Netchip NET2280 target controller driver 20
networking components 19
node IDs 47

O

OHCI 3, 31
 host controller initialization 16
 specification 7
OHCI host controller driver 12
Once 40
Online support site 9
Open Host Controller Interface, *see* OHCI
OSAL
 definition 34
OUT endpoint
 glossary definition 171

P

- pActLen
 - descriptorGet() 117
 - descriptorSet() 117
- pAlternateSetting
 - interfaceGet() 117
- param
 - callback routine parameter 106
- pBfr 118
 - descriptorGet() 116
 - statusGet() 118
 - USB_BFR_LIST 129
- pBuf
 - usbTargPipeStatusGet() 130
- pBuffer
 - copying data to and from 145
 - TRB_COPY_DATA_TO/FROM_EPBUF 145
- pciFindDevice()
 - code example 100
- pConfiguration 113
- pContext
 - mngmtFunc() for TARG_MNGMT_ATTACH 108
 - mngmtFunc() for TARG_MNGMT_BUS_RESET 109
 - mngmtFunc() for TARG_MNGMT_DETACH 109
 - mngmtFunc() for TARG_MNGMT_DISCONNECT 110
 - mngmtFunc() for TARG_MNGMT_SUSPEND and TARG_MNGMT_RESUME 110
 - mngmtFunc() parameter 108
- pContext parameter
 - mngmtFunc() parameter described 108
- pcPentium 26
- pDeviceInfo
 - TRB_ATTACH 140
- PDIUSB12 Evaluation Board (PC Kit) User's Manual 8
- pegasus
 - configuration parameters 15
- Pegasus communication class driver 13, 14
- Pegasus END driver initialization 19
- Pegasus Ethernet interface 19
- Pegasus IP Address 15
- Pegasus Net Mask 15
- Pegasus network device
 - initialization 26
- Pegasus Target Name 15
- PEGASUS_IP_ADDRESS 15, 26
- PEGASUS_MAX_DEVS 15, 26
- PEGASUS_NET_MASK 15, 26
- PEGASUS_TARGET_NAME 15, 26
- pEndpointDesc
 - TRB_ENDPOINT_ASSIGN 143
- per-client data, releasing 38
- peripheral
 - glossary definition 171
- peripheral device components 17
- peripheral stack, *see* USB peripheral stack
- pErp
 - usbTargTransferAbort() 129
- pFrameNo
 - synchFrameGet() 119
 - usbTargCurrentFrameGet() 131
- pHalDeviceInfo 139
 - TRB_ATTACH 140
- pHalPipeHandle 121
- Philips ISP1582 target controller driver 20
- Philips PDIUSB12 target controller driver 20
- Philips USB programming guide 8
- pid
 - USB_BFR_LIST 129
- pipe
 - data transfer 53
 - glossary definition 171
- pipe handle
 - glossary definition 171
- pipeHandle 121, 125, 143, 145
 - TRB_COPY_DATA_TO/FROM_EPBUF 145
 - TRB_ENDPOINT_ASSIGN 143
 - TRB_ENDPOINT_RELEASE 143
 - TRB_IS_BUFFER_EMPTY 145
 - usbTargPipeDestroy() 124
- Platform getting started guide 8, 9, 11, 18
- Platform release notes 27
- play 157
- polling
 - glossary definition 172

- polling interval 124
- pollInput() 69
- power management 2
- pPipeHandle
 - USB_ERP 128
 - usbTargPipeCreate() 121
- print 157
- print command
 - printer application 161
- print4k 157
 - example 160, 161
- printer
 - application testing 159
 - initialization 25
- printer class driver 13
- printer driver 77
 - dynamic attachment 79
 - IEEE-1284 80
 - initialization 77
 - ioctl routines 80
- printer emulator 21
- printers
 - data flow 80
- prnDown 157
- prnDump 161
 - command from target peripheral 160
 - example 160
- prnInit 157
- protocol
 - glossary definition 172
- pTargChannel
 - usbTargTcdAttach() 106
- pTargTcd 121
- pTrb
 - single entry point 136
- pUSB_IRP 96

R

- rebuilding source
 - configuration options 18
- reference pages 7
- registering client modules 36, 43
 - USBHST_DEVICE_DRIVER data structure 36

- registering the host controller driver 54
- registering the hub class driver 62
- release
 - glossary definition 172
- Remote Wakeup
 - glossary definition 172
 - setting and clearing a device 132
 - supporting 139
 - uFeatureSelector element 142
- removing a device from a hub 64
- request
 - vendorSpecific() 120
- request types
 - table of standard 111
- requestType 115
 - descriptorGet() 114, 115
 - descriptorSet() 114, 115
 - featureClear() 111
 - featureSet() 111
 - statusGet() 118, 119
 - vendorSpecific() 120
- reset
 - glossary definition 172
- reset event 148
- result
 - USB_ERP 127, 128, 129
- resume
 - glossary definition 172
- resume event 148
- resume signal 107
- RESUME signaling 131
- root hub
 - glossary definition 172
- root hub emulation 60
- root port
 - glossary definition 172

S

- S_usbTcdLib_BAD_HANDLE 127
- S_usbTcdLib_BAD_PARAM 127
- S_usbTcdLib_CANNOT_CANCEL 127
- S_usbTcdLib_COMM_FAULT 127
- S_usbTcdLib_DATA_OVERRUN 127

- S_usbTcdLib_DATA_TOGGLE_FAULT 127
- S_usbTcdLib_ERP_CANCELLED 127
 - aborting a transfer 130
- S_usbTcdLib_GENERAL_FAULT 127
- S_usbTcdLib_HW_NOT_READY 127
- S_usbTcdLib_INT_HOOK_FAILED 127
- S_usbTcdLib_NEW_SETUP_PACKET 127
- S_usbTcdLib_NOT_IMPLEMENTED 127
- S_usbTcdLib_NOT_INITIALIZED 127
- S_usbTcdLib_NOT_SUPPORTED 127
- S_usbTcdLib_OUT_OF_MEMORY 127
- S_usbTcdLib_OUT_OF_RESOURCES 127
- S_usbTcdLib_PID_MISMATCH 127
- S_usbTcdLib_SHUTDOWN 127
- S_usbTcdLib_STALL_ERROR 127
- sample code
 - usbPciConfigHeaderGet() 100
- SCSI Primary Commands
 - 2 (SPC-2), Revision 3 86
- SCSI-10 commands 26
- SCSI-6 commands 26
- sd_seqRd 86
- sd_seqWrt() 85
- select feature 19
- SEQ_DEV 81, 83, 84, 85
- SEQ_DEV driver model 81
- SET_ADDRESS 119, 141
 - request 60
 - table of control requests 107
- SET_CONFIGURATION 39
 - request 60
 - request, response to 114
 - table of control requests 106
- SET_DESCRIPTOR 39, 114, 117
 - table of control requests 106
- SET_FEATURE 40, 110, 132, 141
 - featureSet() callback 110
 - request 60
 - table of control requests 106
- SET_INTERFACE 39, 107, 117
- setBitRate() 57
- setting and clearing a device
 - Remote Wakeup 132
- setup
 - glossary definition 172

- setup packet
 - glossary definition 172
- single entry point 136
- SIO driver model 66
- SIO_CALLBACK_PUT_MOUSE_REPORT 76
- SIO_CALLBACK_PUT_RCV_CHAR 76
- SIO_CHAN 68, 69, 75, 79, 80
- SIO_MODE_INT 19, 80
- SOF
 - glossary definition 173
- speaker
 - initialization 25
- Speaker audio demonstration configlet 14
- speaker driver 13
 - dynamic attachment 83
 - initialization 81
 - overview 80
 - SEQ_DEV driver model 81
- speakers 80
 - data flow 85
 - recognizing and handling 84
- specifications
 - EHCI 7
 - host controller 3
 - OHCI 3, 7
 - UHCI 7
- spkrDown 157
- spkrFmt 157
- spkrInit 157
- split errors
 - handling 58, 59
- stall
 - generic endpoints 130
 - glossary definition 173
- STALL state 130
- standard request
 - glossary definition 173
 - interfaces 39
- standard request interfaces 39
- start-of-frame, *see* SOF
- state
 - glossary definition 173
- status
 - glossary definition 173

statusGet() 103, 119
 described 118
 table of associated control requests 107
submitURB() 56
suspend
 glossary definition 173
suspend event 148
suspend signal 107
SUSPEND state 131, 141
symmetric multiprocessing 6
SYNCH_FRAME 40, 120
synchFrameGet 107
synchFrameGet() 103, 120
 table of associated control requests 107
synchronization type 123

T

TARG_MNGMT_ATTACH
 management code parameters 107
 usage described 108
TARG_MNGMT_BUS_RESET
 management code parameters 107
 management event code 109
TARG_MNGMT_DETACH 108
 management code parameters 107
 usage described 109
TARG_MNGMT_DISCONNECT
 management code parameters 107
 usage described 110
TARG_MNGMT_RESUME 110
 management code parameters 107
 usage described 110
TARG_MNGMT_SUB_RESET
 USB_TCD_FULL_SPEED, parameter to 109
 USB_TCD_HIGH_SPEED, parameter to 109
 USB_TCD_LOW_SPEED, parameter to 109
TARG_MNGMT_SUSPEND
 management code parameters 107
 usage described 110
TARG_PIPE 121
TARG_TCD 126
targCallback() 128
 USB_ERP 128
targChannel, callback routine parameter 106
target 173
target application 6
 attaching to TCD 5
 callback table 102
 glossary definition 173
target architectures 6
target channel
 glossary definition 173
target controller 4
 glossary definition 173
target controller driver, *see* TCD
target layer
 diagram 98
 functions of 5
 glossary definition 174
 initializing 99
 overview 5
 uninitializing 99
 usage count 99
target request block, *see* TRB
targInit kbd
 example 158
 usage 157
targInit prn
 example 159
 usage 159
TargLib 121
targLink
 USB_ERP 126
targPtr
 USB_ERP 126
TC
 glossary definition 173
TCD 4
 attaching to target application 5
 detaching 101
 disabling 104
 enabling 104
 enabling and disabling 104
 initialization 5
 mass storage TA attachment 103
 overview of responsibilities 4
 single entry point 136
 Target Controller Driver, abbreviation for 4

- TCD handle
 - USB_TARG_CHANNEL, stored in 99
 - TCD_ENDPOINT_STALL 130, 144
 - TCD_ENDPOINT_UNSTALL 130, 144
 - TCD_FNC_ADDRESS_SET 137, 141
 - TCD_FNC_ATTACH 101, 137, 138, 139
 - TCD_FNC_COPY_DATA_FROM_EPBUF 129, 137, 144
 - TCD_FNC_COPY_DATA_TO_EPBUF 129, 137, 144
 - TCD_FNC_CURRENT_FRAME_GET 131, 137, 142
 - TCD_FNC_DETACH 101, 137, 140
 - TCD_FNC_DEVICE_FEATURE_CLEAR 137, 141
 - calling the TCD entry point with 132
 - TCD_FNC_DEVICE_FEATURE_SET 137, 141
 - calling entry point of TCD with 132
 - TCD_FNC_DISABLE 104, 137, 140
 - TCD_FNC_ENABLE 104, 137, 140
 - TCD_FNC_ENDPOINT_ASSIGN 122, 137, 143
 - TCD_FNC_ENDPOINT_INTERRUPT_STATUS_CLEAR 138, 148
 - TCD_FNC_ENDPOINT_INTERRUPT_STATUS_GET 138, 147
 - TCD_FNC_ENDPOINT_RELEASE 124, 137, 143
 - TCD_FNC_ENDPOINT_STATE_SET 130, 137, 143
 - TCD_FNC_ENDPOINT_STATUS_GET 130, 137, 144
 - TCD_FNC_HANDLE_DISCONNECT_INTERRUPT 138, 148
 - TCD_FNC_HANDLE_RESET_INTERRUPT 138, 148
 - TCD_FNC_HANDLE_RESUME_INTERRUPT 138, 148, 149
 - TCD_FNC_HANDLE_SUSPEND_INTERRUPT 138, 148, 149
 - TCD_FNC_INTERRUPT_STATUS_CLEAR 138, 145, 146
 - TCD_FNC_INTERRUPT_STATUS_GET 138, 145, 147
 - TCD_FNC_IS_BUFFER_EMPTY 137, 145
 - TCD_FNC_RESUME 141
 - TCD_FNC_SIGNAL_RESUME 131, 137
 - TCD-defined parameters 100
 - tcdLink
 - USB_ERP 126
 - tcdParam
 - TRB_ATTACH 139
 - tcdPtr
 - USB_ERP 126
 - Test Mode
 - feature cannot be cleared 132
 - glossary definition 174
 - setting and clearing a device feature 132
 - uFeatureSelector element specifies the 142
 - whether a device supports 139
 - test selectors 132, 142
 - Testing Tool 22
 - token packet
 - glossary definition 174
 - toolchain
 - glossary definition 174
 - transaction
 - glossary definition 174
 - transfer type 123
 - glossary definition 174
 - transferType
 - USB_ERP 128
 - translation layer 33
 - translation unit
 - glossary definition 175
 - TRB
 - description of 136
 - glossary definition 174
 - header 136
 - specific codes in a 136
 - trbLength
 - TRB header 138
 - tUsbCln 44
 - tUsbKbd 72
 - tUsbMSCXbdStrategy 88
 - tUsbMSCXbdStrategy task 88
 - typematic repeat 72
- ## U
- uActLength
 - TRB_COPY_DATA_TO/FROM_EPBUF 145
 - uDeviceFeature
 - USB_APPLN_DEVICE_INFO 139

- USB_APPLN_DEVICE_INFO structure
 - member 108
- uEndpointInterruptStatus
 - TRB_ENDPOINT_INTERRUPT_STATUS_GET 147
- uEndpointNumberBitmap
 - USB_APPLN_DEVICE_INFO 108, 139
- uFeatureSelector
 - TRB_DEVICE_FEATURE_SET_CLEAR 142
 - usbTargDeviceFeatureSet() 132
- UFI_MAX_DEVS 15
- UFI_MAX_DRV_NAME_SZ 15
- UHCI 3, 31
 - glossary definition 175
 - host controller initialization 16
 - specification 7
- UHCI host controller driver 12, 13
- uInterruptStatus 145
 - TRB_INTERRUPT_STATUS_GET_CLEAR 145
- Universal Host Controller Interface, *see* UHCI
- Universal Serial Bus Driver, *see* USBD
- Universal Serial Bus Specification 7, 8
- un-stall 130
- uNumberEndpoints
 - USBHAL_DEVICE_INFO 139
- URB
 - glossary definition 175
- usage type 123
- USB 1.0 compatibility 4
- USB 2.0
 - technology overview 2
- USB 2.0 compatibility 4
- USB 2.0 host controller driver 34
- USB 2.0 hub class driver 33
- USB 2.0 translation layer 33
- USB 2.0 USBD layer 33
- USB Audio Demo components 27
- USB Bulk Device Name Size 14
- USB Bulk Drive Name 14
- USB Bulk Maximum Drives 14
- USB CBI Drive Name 15
- USB class drivers 32
- USB classes 44
- USB data toggles 49
- USB Device Class Definition for Human Interface Devices 7
- USB Device Class Definition for Printing Devices 7
- USB device class specifications 7
- USB devices
 - glossary definition 175
 - initialization 17
- USB Floppy Interface (UFI) Command Specification 86
- USB host controller
 - drivers 31
 - overview 31
- USB host driver 31, 33
 - see also* USB host stack
 - USB 1.0 compatibility 43
 - USB 4
- USB host modules 32
- USB host stack 20
 - architecture 29
 - component dependencies 18
 - configuration 8
 - configuring and building VxWorks 11
 - general component dependencies 18
 - initialization 16, 22
 - overview 3
- USB host stack architecture, overview 29
- USB initialization process 23
- USB Mass Storage Class Bulk Only Transport Specification 7
- USB Mass Storage Class Control/Bulk/Interrupt (CBI) Transport Specification 7
- USB Mass Storage Class Specification 7
- USB Mass Storage Class UFI Command Specification 7
- USB Pegasus Device Maximum Number 15
- USB Pegasus END driver 19
- USB peripheral stack
 - glossary definition 171
 - including the component 21
- USB peripheral stack components 21
- USB protocols 44
- USB Request Block, *see* URB
- USB speakers
 - compound devices 81
- USB subclasses 44

- USB UFI Device Name Size 15
- usb.h 44, 51
- USB_APPLN_DEVICE_INFO
 - structure
 - elements 108
 - mngmtFunc(), parameter to 108
 - TCD populating 139
- USB_BFR_LIST 128, 129
- USB_BULK_NON_REMOVABLE_DISK 14, 15
- USB_DATA0 128
- USB_DATA1 128
- USB_DESCR_CONFIGURATION 116
- USB_DESCR_DEVICE 116
- USB_DESCR_DEVICE_QUALIFIER 116
- USB_DESCR_ENDPOINT 116
 - descriptorType 122
- USB_DESCR_INTERFACE 116
- USB_DESCR_INTERFACE_POWER 116
- USB_DESCR_OTHER_SPEED_CONFIGURATION 116
- USB_DESCR_STRING 116
- USB_DEV_STS_LOCAL_POWER 118, 119
- USB_DEV_STS_REMOTE_WAKEUP 118, 119
- USB_ENDPOINT_DESCR 122
- USB_ENDPOINT_STS_HALT 119, 130
- USB_ERP 126
 - size 126
 - structure 125
- USB_FEATURE_DEVICE_REMOTE_WAKEUP 108, 139
- USB_FEATURE_TEST_MODE 108, 139
- USB_FEATURE_USB20 108, 139
- USB_FSEL_DEV_ENDPOINT_HALT 113
- USB_FSEL_DEV_REMOTE_WAKEUP 113
- USB_FSEL_DEV_TEST_MODE 113
- USB_HAL_PIPE_INFO 121
- USB_IRP 51
- USB_KBD_ATTACH 68
- USB_KBD_QUEUE_SIZE 16
- USB_KBD_REMOVE 68
- USB_MAX_KEYBOARDS 16
- USB_MSC_ATTACH 91
- USB_MSC_DETACH 91
- USB_MSE_ATTACH 75
- USB_MSE_REMOVE 75
- USB_NODE_ID 47, 91
- USB_PID_IN 129
- USB_PID_OUT 129
- USB_PID_SETUP 129
- USB_PRN_ATTACH 79
- USB_PRN_REMOVE 79
- USB_REQ_CLEAR_FEATURE 111, 112
- USB_REQ_GET_CONFIGURATION 111
- USB_REQ_GET_DESCRIPTOR 111
- USB_REQ_GET_INTERFACE 111
- USB_REQ_GET_STATE 111
- USB_REQ_GET_STATUS 111
- USB_REQ_SET_ADDRESS 111
- USB_REQ_SET_CONFIGURATION 111
- USB_REQ_SET_DESCRIPTOR 111
- USB_REQ_SET_FEATURE 111
- USB_REQ_SET_FEATURE Request, described 112
- USB_REQ_SET_INTERFACE 111
- USB_REQ_SYNCH_FRAME 111
- USB_RT_CLASS 115
- USB_RT_DEV_TO_HOST 115
- USB_RT_DEVICE 115
- USB_RT_ENDPOINT 115
- USB_RT_HOST_TO_DEV 115
- USB_RT_INTERFACE 115
- USB_RT_OTHER 115
- USB_RT_STANDARD 115
- USB_RT_VENDOR 115
- USB_SCSI_FLAG_READ_WRITE10 26
- USB_SCSI_FLAG_READ_WRITE6 26
- USB_SPKR_ATTACH 83
- USB_SPKR_REMOVE 83
- USB_TARG_ADDRESS_SET_FUNC 103
- USB_TARG_CHANNEL
 - parameter to callback routines 106
 - storing the TCD handle 99
- USB_TARG_CONFIGURATION_GET_FUNC 102
- USB_TARG_CONFIGURATION_SET_FUNC 103
- USB_TARG_DESCRIPTOR_GET_FUNC 103
- USB_TARG_DESCRIPTOR_SET_FUNC 103
- USB_TARG_FEATURE_CLEAR_FUNC 102
- USB_TARG_FEATURE_SET_FUNC 102
- USB_TARG_INTERFACE_GET_FUNC 103
- USB_TARG_INTERFACE_SET_FUNC 103
- USB_TARG_MANAGEMENT_FUNC 102

- USB_TARG_STATUS_GET_FUNC 103
- USB_TARG_SYNCH_FRAME_GET_FUNC 103
- USB_TARG_VENDOR_SPECIFIC_FUNC 103
- USB_TCD_FULL_SPEED
 - TARG_MNGMT_BUS_RESET 109
- USB_TCD_HIGH_SPEED
 - TARG_MNGMT_BUS_RESET 109
- USB_TCD_LOW_SPEED
 - TARG_MNGMT_BUS_RESET 109
- USB_TEST_MODE_J_STATE 132
- USB_TEST_MODE_K_STATE 132
- USB_TEST_MODE_SE0_ACK 132
- USB_TEST_MODE_TEST_FORCE_ENABLE 132
- USB_TEST_MODE_TEST_PACKET 132
- USB_XFRTYPE_BULK 128
- USB_XFRTYPE_CONTROL 128
- USB_XFRTYPE_INTERRUPT 128
- USB_XFRTYPE_ISOCH 128
- USB_XXX_ATTACH 94
- USB_XXX_DETACH 94
- USB_XXX_DEV 94
- usbAcmLib 33
- usbAudio 26, 27
- usbAudio demo
 - initialization 26
- usbBrcmAudioMicrophoneTest 20
- usbBrcmAudioSpkrTest 20
- usbBrcmAudioTestStop() 20
- usbBulkDevLib 33
- usbCbiUfiDevLib 33
- USB_D 12
 - glossary definition 175
 - initialization 23, 34
- USB_D 1.1 43
 - compatibility 42
- USB_D 2.0 34
- USB_D Client registration
 - USB_D Client deregistration 43
- USB_D, *see* USB host driver
- USB_D_CLIENT_HANDLE 43, 67
- USB_D_PIPE_HANDLE 50
- usbDbusCountGet() 48
- usbDbclientRegister() 43, 44, 67
- usbDbclientUnregister() 43, 68
- usbDbconfigurationSet() 49
- usbDbdynamicAttachRegister() 44, 67
- usbDbhubPortCountGet() 48
- usbDbinit() 35
- usbDbinit() 35
- usbDbinitialize() 43, 91
- usbDbLib 48
- usbDbLib.h 45
- usbDbnodeIdGet() 48
- usbDbnodeInfoGet() 48
- usbDbrootNodeIdGet() 48
- usbDbtargTransfer() 125
- usbDbtransfer() 53
- usbDbtransferAbort() 53
- usbEnum
 - command at usbTool prompt 158
 - example 158, 160
- USBHAL_DEVICE_INFO 139
- USBHAL_PIPE_INFO 142
- usbHallSr() 139
- usbHallSrParam 139
 - TRB_ATTACH 139
- usbHcdInit() 35
- usbHid.h 76
- usbHst.h 40
- USBHST_BAD_START_OF_FRAME 60
- USBHST_BUFFER_OVERRUN_ERROR 60
- USBHST_BUFFER_UNDERRUN_ERROR 60
- USBHST_DATA_OVERRUN_ERROR 60
- USBHST_DATA_UNDERRUN_ERROR 60
- USBHST_DEVICE_DRIVER
 - data structure defined 36
- USBHST_DEVICE_NOT_RESPONDING_ERROR 60
- USBHST_HC_DRIVER 54
- USBHST_HC_DRIVER Structure 56
- USBHST_INSUFFICIENT_BANDWIDTH 59
- USBHST_INSUFFICIENT_RESOURCE 59
- USBHST_INVALID_PARAMETER 59
- USBHST_MEMORY_NOT_ALLOCATED 59
- USBHST_STALL_ERROR 60
- USBHST_TIMEOUT 60
- USBHST_TRANSFER_CANCELLED 60
- USBHST_URB 40
- USBHST_USB_D_TO_HCD_FUNCTION_LIST 57
 - Structure 57

- usbHstBusDeregister() 59
- usbHstBusRegister() 59
- usbHstClearFeature() 39
- usbHstDriverDeregister() 38, 65
- usbHstDriverRegister() 37, 62
- usbHstGetConfiguration() 39
- usbHstGetDescriptor() 39
- usbHstGetInterface() 39
- usbHstGetStatus() 40
- usbHstHCDDeregister() 59
- usbHstHCDRegister() 54, 59
- usbHstSetConfiguration() 39
- usbHstSetDescriptor()
 - description 39
 - example 40
- usbHstSetFeature() 40
- usbHstSetInterface() 39
- usbHstSetSynchFrame() 40
- usbHstURBCancel() 41
- usbHstURBSubmit()
 - data transfer interface 41
 - example 41
- usbHubExit() 62
- usbHubInit() 35
- usbHubInit() 62
- usbi 160
- usbInit
 - command for usbTool 156
- usbInit() 23
- usbInit() configlet 23
- usbKeyboardDevInit() 66
- usbKeyboardDynamicAttachRegister() 68
- usbKeyboardLib 33, 65, 66, 68, 69, 75
- usbKeyboardSioChanLock() 69
- usbKeyboardSioChanUnlock() 69
- usbMouseDevInit() 73
- usbMouseDevShutdown() 76
- usbMouseDynamicAttachRegister() 75
- usbMouseLib 33, 73, 75, 76
- usbMouseSioChanLock() 75
- usbMouseSioChanUnlock() 75
- usbMSCBlkDevCreate() 25, 26, 91
 - allocating an XBD 88
- usbMSCBlkRd() 90
- usbMSCBlkWrt() 90
- usbMSCDevCreate() 90
- usbMSCDevInit() 90, 91
- usbMSCDevIoctl() 90
- usbMSCDevLock() 91
- usbMSCDevUnlock() 91
- usbMSCDynamicAttachRegister() 91
- usbMSCDynamicAttachUnregister() 91
- usbMSCIrpCallback() 92
- usbMSCStatusChk() 90
- usbMSCXbdStrategy 88
- usbPciConfigHeaderGet() 100
 - sample code 100
- usbPegasusEndLib 33
- usbPrinterDevInit() 77
- usbPrinterDynamicAttachRegister() 79
- usbPrinterLib 33, 77, 79, 80
- usbPrinterSioChanLock() 79
- usbPrinterSioChanUnlock() 80
- usbSpeakerDevInit() 81
- usbSpeakerDynamicAttachRegister() 83, 84
- usbSpeakerLib 33, 80, 81, 83, 84, 85, 86
- usbSpeakerSeqDevLock() 83, 85
- usbSpeakerSeqDevUnlock() 83, 85
- usbTargControlPayloadRcv() 120, 122, 125, 130, 131
- usbTargControlResponseSend() 120, 122, 125, 130, 131
- usbTargControlStatusSend() 120, 122, 125, 130, 131
- usbTargCurrentFrameGet() 131
- usbTargDeviceFeatureClear() 112, 132
- usbTargDeviceFeatureSet() 112, 132
- usbTargDisable() 105
 - enabling and disabling the TCD 104
- usbTargEnable() 99, 104, 105
 - attaching a TCD 100
 - enabling and disabling the TCD 104
- usbTargInitialize() 99, 133
- usbTargLib.h 107
 - callback table defined in 106
- usbTargPipeCreate() 121, 122, 124
- usbTargPipeDestroy() 121, 124
- usbTargPipeStatusGet() 130
- usbTargPipeStatusSet() 112, 130
- usbTargShutdown() 99, 133

- usbTargSignalResume() 131
- usbTargStatusGet() 130
- usbTargTcdAttach() 99, 101, 103, 104, 105, 139
 - enabling and disabling the TCD 104
 - establishing communication 101
 - prototype 99
- usbTargTcdDetach() 101
 - prototype 101
- usbTargTransfer() 124
 - aborting a transfer submitted by 129
 - transferring data 124
- usbTargTransferAbort() 129
 - prototype 129
- usbTcd.h 147
- USBTCDC_ENDPOINT_BIT_STUFF_ERROR 147
- USBTCDC_ENDPOINT_COMMUN_FAULT 147
- USBTCDC_ENDPOINT_CRC_ERROR 147
- USBTCDC_ENDPOINT_DATA_OVERRUN 147
- USBTCDC_ENDPOINT_DATA_TOGGLE_ERROR 147
- USBTCDC_ENDPOINT_DATA_UNDERRUN 147
- USBTCDC_ENDPOINT_PID_MISMATCH 147
- USBTCDC_ENDPOINT_STALL_ERROR 147
- USBTCDC_ENDPOINT_TIMEOUT_ERROR 147
- USBTCDC_ENDPOINT_TRANSFER_SUCCESS 147
- usbTool 22, 32, 155
 - attach ehci 156
 - attach ohci 156
 - attach uhci 156
 - caution with initialization components 24
 - code exerciser utility 155
 - commands 156
 - execution sequence 156
 - keyboard test commands 157
 - mouse test commands 157
 - printer test commands 157
 - relationship to other modules 32
 - running from the shell 156
 - speaker test commands 157
 - testing devices 156
 - testing the keyboard application with 157
 - testing the printer application with 159
 - usbInit 156
- usbTool Test Application
 - using 155

- usbTool() 32
- usbXXXDevLock() 94
- usbXXXDevUnlock() 95
- usbXXXDynamicAttachRegister() 94
- usbXXXEndInit() 95
- usbXXXEndLib 92
- userCallback() 127, 128
 - aborting a transfer 129
 - required for result field 127
 - USB_BRF_LIST structure, actLen 129
 - USB_ERP 128
- userPtr, USB_ERP 126
- usrUsbBulkDevInit.c 25
- usrUsbPegasusEndInit.c
 - initializing Pegasus class driver 26
- usrUsbTool.c 32
- uTestSelector 142
 - TRB_DEVICE_FEATURE_SET_CLEAR 142
 - usbTargDeviceFeatureSet() 132
- utilities, usbTool 155

V

- value, vendorSpecific() 120
- Vendor-specific devices 38
- vendorSpecific() 103, 107, 120
 - table of associated control requests 107
- volume 157
- vxbUsbControllerRegister() 34
- VxWorks
 - component dependencies 18
 - configuring components 11
 - task name, example 36

W

- wakeup
 - glossary definition 175
- wav file 19
- Wind River Online Support 9
- wIndex, vendorSpecific() 120
- wLength 118

statusGet() [118](#)
 vendorSpecific() [120](#)
wValue [120](#)

X

XBD [25](#)
 device driver model
 driver model [88](#)
 structures [88, 91](#)
XXXEndRecv() [96](#)
XXXEndSend() [96](#)

Z

zero-length packet [126](#)