

Wind River[®] USB for VxWorks[®] 6

API REFERENCE

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, the Wind River logo, Tornado, and VxWorks are registered trademarks 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. Libraries

This section provides reference entries for each of the Wind River USB libraries, arranged alphabetically. Each entry lists the routines found in the library, including a one-line synopsis of each and a general description of their use.

Individual reference entries for each of the available functions in these libraries is provided in section 2.

2. Routines

This section provides reference entries for each of the routines found in the Wind River USB libraries documented in section 1.

1

Libraries

cmdParser	– Command line parser routines.	3
ossLib	– O/S-independent services for vxWorks	4
usbBulkDevLib	– USB Bulk Only Mass Storage class driver	5
usbCbiUfiDevLib	– USB CBI Mass Storage class driver for UFI sub-class	7
usbDescrCopyLib	– USB descriptor copy utility functions	8
usbEhcdBandwidth	– contains the bandwidth functions of EHCD	9
usbEhcdEventHandler	– USB EHCI HCD interrupt handler	9
usbEhcdInitExit	– USB EHCI HCD initialization routine	9
usbEhcdRhEmulation	– USB EHCI HCD RootHub Emulation	10
usbEhcdTransferManagement	– transfer management functions of the EHCD	10
usbEhcdUtil	– contains the utility functions of EHCD	11
usbHalDeviceControlStatus	– HAL Device Control and Status handler module	11
usbHalEndpoint	– HAL Endpoint specific functionalities	11
usbHalInitExit	– HAL initialization and uninitialization functionalities	12
usbHalInterruptHandler	– USB HAL interrupt handler module	12
usbHalUtil	– Utility functions of HAL	13
usbHandleLib	– handle utility functions	13
usbHubInitialization	– Initialization and cleanup of HUB class driver	13
usbKeyboardLib	– USB keyboard class drive with vxWorks SIO interface	14
usbLib	– USB utility functions	16
usbListLib	– Linked list utility functions	16
usbMouseLib	– USB mouse class drive with vxWorks SIO interface	17
usbOhci	– USB OHCI Driver Entry and Exit points	19
usbOhciDebug	– USB OHCI Debug Routines	19
usbPegasusEnd	– USB Ethernet driver for the Pegasus USB-Ethernet adapter	19
usbPrinterLib	– USB printer class drive with vxWorks SIO interface	22
usbQueueLib	– O/S-independent queue functions	23
usbSpeakerLib	– USB speaker class drive with vxWorks SEQ_DEV interface	24
usbTargDefaultPipe	– Handles the requests to the default control pipe	27
usbTargDeviceControl	– modules for handling pipe specific requests	27

usbTargInitExit	– USB Initialization/Uninitialization modules	28
usbTargKbdLib	– USB keyboard target exerciser/demonstration	28
usbTargMsLib	– Mass Storage routine library	29
usbTargPipeFunc	– modules for handling pipe specific requests	30
usbTargPrnLib	– USB printer target exerciser/demonstration	30
usbTargRbcCmd	– Reduced Block Command set routine library	31
usbTargRbcLib	– USB Reduced Block Command set routine library	32
usbTargUtil	– Utility Functions	32
usbTcdIsp1582InitExit	– Initialization/uninitialization for ISP 1582 TCD	33
usbTcdNET2280InitExit	– initialization/uninitialization for NET2280 TCD	33
usbTcdPdiusb12InitExit	– Initialization/uninitialization for PDIUSB12 TCD	34
usbTransUnitData	– Translation Unit Data Transfer Interfaces	34
usbTransUnitInit	– Translation Unit Initialization interfaces	35
usbTransUnitMisc	– translation unit miscellaneous functions	36
usbTransUnitStd	– translation unit standard requests interfaces	36
usbUhdInitialization	– USB UHCI HCD initialization routine	37
usbUhdIsr	– USB UHCI HCD interrupt handler	37
usbUhdManagePort	– USB UHCI HCD port status handler	38
usbUhdRhEmulate	– USB UHCI HCD RootHub Emulation	38
usbUhdScheduleQSupport	– USB UHCD HCD schedule queue support	38
usbUhdScheduleQWaitForSignal	– USB UHCD HCD ISR support routines	39
usbUhdScheduleQueue	– USB UHCD HCD schedule queue routines	39
usbUhdSupport	– USB UHCD HCD register access routines	39
usbVxbRegAccess	– library for read/write routines	40
usbdb	– USBDB Routines	40

cmdParser

NAME `cmdParser` – Command line parser routines.

ROUTINES

- `PromptAndExecCmd()` – Prompt for a command and execute it.
- `KeywordMatch()` – Compare keywords
- `ExecCmd()` – Execute the command line
- `SkipSpace()` – Skips leading white space in a string
- `TruncSpace()` – Truncates string to eliminate trailing whitespace
- `GetNextToken()` – Retrieves the next token from an input string
- `GetHexToken()` – Retrieves value of hex token
- `CmdParserHelpFunc()` – Displays list of supported commands
- `CmdParserExitFunc()` – Terminates parser execution

DESCRIPTION This file includes a collection of command-line parsing functions which are useful in the creation of command line utilities, such as bus exercisers.

There are three groups of functions defined by this library. The first is a collection of general string parser functions, such as functions to eliminate white space, functions to strip tokens out of a string, and so forth.

The second set of functions drive the actual parsing process. In order to use this second set of functions, clients must construct a table of `CMD_DESCR` structures which define the commands to be recognized by the parser. A brief example of such a table is shown below.

```
CMD_DESCR Commands [] =
{
    {"Help", 4, "Help/?", "Displays list of commands.", CmdParserHelpFunc},
    {"?", 1, NULL, NULL, CmdParserHelpFunc},
    {"Exit", 4, "Exit", "Exits program.", CmdParserExitFunc},
    {NULL, 0, NULL, NULL, NULL}
};
```

The first field is the keyword for the command. The second field specifies the number of characters of the command which must match - allowing the user to enter only a portion of the keyword as a shortcut. The third and fourth fields are strings giving the command usage and a brief help string. A `NULL` in the Help field indicates that the corresponding keyword is a synonym for another command its usage/help should not be shown. The final field is a pointer to a function of type `CMD_EXEC_FUNC` which will be invoked if the parser encounters the corresponding command.

The third group of functions provide standard `CMD_EXEC_FUNCs` for certain commonly used commands, such as `CmdParserHelpFunc` and `CmdParserExitFunc` as shown in the preceding example.

The caller may pass a generic (`pVOID`) parameter to the command line parsing functions in the second group. This function is in turn passed to the `CMD_EXEC_FUNCs`. In this way, the caller can specify context information for the command execution functions.

Commands are executed after the user presses [enter]. Multiple commands may be entered on the same command line separated by semicolons (;). Each command as if it had been entered on a separate line (unless a command terminates with an error, in which case all remaining commands entered on the same line will be ignored).

INCLUDE FILES **cmdParser.h**

ossLib

NAME **ossLib** – O/S-independent services for vxWorks

ROUTINES **ossStatus()** – Returns **OK** or **ERROR** and sets **errno** based on status.
ossShutdown() – Shuts down **ossLib**.
ossThreadCreate() – Spawns a new thread.
ossThreadDestroy() – Attempts to destroy a thread.
ossThreadSleep() – Voluntarily relinquishes the CPU.
ossSemCreate() – Creates a new semaphore.
ossSemDestroy() – Destroys a semaphore.
ossSemGive() – Signals a semaphore.
ossSemTake() – Attempts to take a semaphore.
ossMutexCreate() – Creates a new mutex.
ossMutexDestroy() – Destroys a mutex.
ossMutexTake() – Attempts to take a mutex.
ossMutexRelease() – Releases (gives) a mutex.
ossPartSizeGet() – Retrieves the size of the USB memory partition.
ossPartSizeSet() – Sets the the initial size of the USB memory partition.
ossPartIdGet() – Retrieves the partition ID of USB memory partition.
ossMemUsedGet() – Retrieves the amount of memory currently in use by USB.
ossMalloc() – Master USB memory allocation routine.
ossPartMalloc() – USB memory allocation
ossOldMalloc() – Global memory allocation
ossCalloc() – Allocates memory initialized to zeros.
ossFree() – Master USB memory free routine.
ossPartFree() – Frees globally allocated memory.
ossOldFree() – Frees globally allocated memory.
ossOldInstall() – Installs the old method of USB malloc and free.
ossTime() – Returns the relative system time in msec.
ossInitialize() – Initializes **ossLib**.

DESCRIPTION Implements functions defined by **ossLib.h**. See **ossLib.h** for a complete description of these functions.

INCLUDE FILES ossLib.h

usbBulkDevLib

NAME **usbBulkDevLib** – USB Bulk Only Mass Storage class driver

ROUTINES **usbBulkDevShutDown()** – shuts down the USB bulk-only class driver
usbBulkDevInit() – registers USB Bulk only mass storage class driver
usbBulkDevIoctl() – perform a device-specific control
usbBulkBlkDevCreate() – create a block device
usbBulkDynamicAttachRegister() – Register SCSI/BULK-ONLY device attach callback.
usbBulkDynamicAttachUnregister() – Unregisters SCSI/BULK-ONLY attach callback.
usbBulkDevLock() – Marks **USB_BULK_DEV** structure as in use
usbBulkDevUnlock() – Marks **USB_BULK_DEV** structure as unused.
usbBulkDriveShow() – shows routine for displaying one LUN of a device.
usbBulkDevShow() – shows routine for displaying all LUNs of a device.
usbBulkShow() – shows routine for displaying all bulk devices.
usbBulkDriveEmpty() – routine to check if drive has media inserted.
usbBulkGetMaxLun() – Return the max LUN number for a device

DESCRIPTION This module implements the USB Mass Storage class driver for the vxWorks operating system. This module presents an interface which is a superset of the vxWorks Block Device driver model. This driver implements external APIs which would be expected of a standard block device driver.

This class driver restricts to Mass Storage class devices that follow bulk-only transport. For bulk-only devices transport of command, data and status occurs solely via bulk endpoints. The default control pipe is only used to set configuration, clear STALL condition on endpoints and to issue class-specific requests.

The class driver is a client of the Universal Serial Bus Driver (USBBD). All interaction with the USB buses and devices is handled through the USBBD.

INITIALIZATION The class driver must be initialized with **usbBulkDevInit()**. It is assumed that USBBD is already initialized and attached to atleast one USB Host Controller. **usbBulkDevInit()** registers the class driver as a client module with USBBD. It also registers a callback routine to get notified whenever a USB MSC/SCSI/ BULK-ONLY device is attached or removed from the system. The callback routine creates a **USB_BULK_DEV** structure to represent the USB device attached. It also sets device configuration, interface settings and creates pipes for **BULK_IN** and **BULK_OUT** transfers.

OTHER FUNCTIONS

usbBulkBlkDevCreate() is the entry point to define a logical block device. This routine initializes the fields within the vxWorks block device structure **XBD**. This **XBD** structure is part of the **USB_BULK_DEV_XBD_LUN** structure. The **USB_BULK_DEV_XBD_LUN** is part of the **USB_BULK_DEV** structure for each of the logical unit in one USB mass storage device. Memory is allocated for **USB_BULK_DEV** by the dynamic attach notification callback routine. So, this create routine just initializes the **XBD** structure and returns a pointer to it, which is used during the file system initialization call.

usbBulkDevIoctl() implements functions which are beyond basic file handling. Class-specific requests, Descriptor show, are some of the functions. Function code parameter identifies the IO operation requested.

DATA FLOW

For each USB MSC/SCSI/BULK-ONLY device detected, **usbBulkPhysDevCreate()** will create pipes to **BULK_IN** and a **BULK_OUT** endpoints of the device. A pipe is a channel between USB client i.e **usbBulkDevLib** and a specific endpoint. All SCSI commands are encapsulated within a Command Block Wrapper (CBW) and transferred across the **BULK_OUT** endpoint through the out pipe created. This is followed by a data transfer phase. Depending on the SCSI command sent to the device, the direction bit in the CBW will indicate whether data is transferred to or from the device. This bit has no significance if no data transfer is expected. Data is transferred to the device through **BULK_OUT** endpoint and if the device is required to transfer data, it does through the **BULK_IN** endpoint. The device shall send Command Status Wrapper (CSW) via **BULK_IN** endpoint. This will indicate the success or failure of the CBW. The data to be transferred to device will be pointed by the file system launched on the device.

INCLUDE FILES

usbBulkDevLib.h

SEE ALSO

"USB Mass Storage Class - Bulk Only Transport Specification Version 1.0, ", "SCSI-2 Standard specification 10L - Direct Access device commands"

usbCbiUfiDevLib

NAME

usbCbiUfiDevLib – USB CBI Mass Storage class driver for UFI sub-class

ROUTINES

usbCbiUfiDevShutDown() – shuts down the USB CBI mass storage class driver
usbCbiUfiDevInit() – registers USB CBI mass storage class driver for UFI devices
usbCbiUfiDevIoctl() – perform a device-specific control.
usbCbiUfiBlkDevCreate() – create a block device
usbCbiUfiDynamicAttachRegister() – Register UFI device attach callback.
usbCbiUfiDynamicAttachUnregister() – Unregisters **CBI_UFI** attach callback.
usbCbiUfiDevLock() – Marks **CBI_UFI_DEV** structure as in use
usbCbiUfiDevUnlock() – Marks **CBI_UFI_DEV** structure as unused.

DESCRIPTION	<p>This module implements the USB Mass Storage class driver for the vxWorks operating system. This module presents an interface which is a superset of the vxWorks Block Device driver model. The driver implements external APIs which would be expected of a standard block device driver.</p> <p>This class driver restricts to Mass Storage class devices with UFI subclass, that follow CBI (Control/Bulk/Interrupt) transport. For CBI devices transport of command, data and status occurs via control, bulk and interrupt endpoints respectively. Interrupt endpoint is used to signal command completion.</p> <p>The class driver is a client of the Universal Serial Bus Driver (USBD). All interaction with the USB buses and devices is handled through the USBD.</p>
INITIALISATION	<p>The driver initialisation routine usbCbiUfiDevInit() must be invoked first prior to any other driver routines. It is assumed that USBD is already initialised and attached to at least one USB Host Controller. usbCbiUfiDevInit() registers the class driver as a client module with USBD. It also registers a callback routine to get notified whenever a USB MSC/UFI/CBI device is attached or removed from the system. The callback routine creates a USB_CBI_UFI_DEV structure to represent the USB device attached. It also sets device configuration, interface settings and creates pipes for BULK_IN, BULK_OUT and INTERRUPT transfers.</p>
DATA FLOW	<p>For every USB/CBI/UFI device detected, the device configuration is set to the configuration that follows the CBI/UFI command set. Pipes are created for bulk in, bulk out and interrupt endpoints. To initiate transactions, ADSC class specific request is used to send a command block on the control endpoint. Command blocks are formed as per the UFI command specifications. If the command requires transport of data to/from the device, it is done via bulk-out/bulk-in pipes using IRPs. This is followed by status transport via interrupt endpoint.</p>
OTHER FUNCTIONS	<p>Number of USB CBI_UFI devices supported by this driver is not fixed. UFI devices may be added or removed from the USB system at any point of time. The user of this client driver must be aware of the device attachment and removal. To facilitate this, an user-specific callback routine may be registered, using usbCbiUfiDynamicAttachRegister() routine. The USBD_NODE_ID assigned to the device being attached or removed, is passed on to the user callback routine. This unique ID may be used to create a block device using usbCbiUfiBlkDevCreate() and further launch file system.</p>
NOTE	<p>The user callback routine is invoked from the USBD client task created for this class driver. The callback routine should not invoke any class driver function, which will further submit IRPs. For example, usbCbiUfiBlkDevCreate() should not be invoked from the user's callback.</p> <p>Typically, the user may create a task, as a client of UFI driver, and invoke the driver routines from the task's context. The user callback routine may be used to notify device attachment and removal to the task.</p>

INCLUDE FILES	usbCbiUfiDevLib.h, blkIo.h
SEE ALSO	<i>USB Mass Storage Class - Control/Bulk/Interrupt Transport Specification Revision 1.0, USB Mass Storage Class - UFI Command specification Revision 1.0</i>

usbDescrCopyLib

NAME	usbDescrCopyLib – USB descriptor copy utility functions
ROUTINES	usbDescrCopy32() – copies descriptor to a buffer usbDescrCopy() – copies descriptor to a buffer usbDescrStrCopy32() – copies an ASCII string to a string descriptor usbDescrStrCopy() – copies an ASCII string to a string descriptor.
DESCRIPTION	This module contains miscellaneous functions which may be used by the USB driver (USBD), USB HCD (USB Host Controller Driver), USB HCD (USB Target Controller Driver) or by USBD clients.
INCLUDE FILES	usbDescrCopyLib.h

usbEhcdBandwidth

NAME	usbEhcdBandwidth – contains the bandwidth functions of EHCD
ROUTINES	
DESCRIPTION	This module defines the bandwidth related functions for the EHCI Host Controller Driver.
INCLUDE FILES	usbOsal.h, usbOsalDebug.h, usbHst.h, usbEhcdDataStructures.h, usbEhcdUtil.h, usbEhcdDebug.h
SEE ALSO	<i>USB specification, revision 2.0, EHCI specification, revision 1.0</i>

usbEhcdEventHandler

NAME	usbEhcdEventHandler – USB EHCI HCD interrupt handler
ROUTINES	
DESCRIPTION	This contains interrupt routines which handle the EHCI interrupts.
INCLUDE FILES	usb2/usbOsal.h, usb2/usbHst.h, usb2/usbEhcdDataStructures.h, usb2/usbEhcdUtil.h, usb2/BusAbstractionLayer.h, usb2/usbEhcdEventHandler.h, usb2/usbEhcdHal.h, usb2/usbEhcdRhEmulation.h, intLib.h

usbEhcdInitExit

NAME	usbEhcdInitExit – USB EHCI HCD initialization routine
ROUTINES	usbEhcdInstantiate() – instantiate the USB EHCI Host Controller Driver. usbEhcdInit() – initializes the EHCI Host Controller Driver usbEhcdExit() – uninitializes the EHCI Host Controller vxbUsbEhciRegister() – registers the EHCI Controller with vxBus
DESCRIPTION	This contains the initialization and uninitialization functions provided by the EHCI Host Controller Driver.
INCLUDE FILES	usb2/usbOsal.h, usb2/BusAbstractionLayer.h, usb2/usbEhcdConfig.h, usb2/usbHst.h, usb2/usbEhcdDataStructures.h, usb2/usbEhcdInterfaces.h, usb2/usbEhcdHal.h, usb2/usbEhcdUtil.h, usb2/usbEhcdEventHandler.h, usb2/usbHcdInstr.h, spinLockLib.h

usbEhcdRhEmulation

NAME	usbEhcdRhEmulation – USB EHCI HCD RootHub Emulation
ROUTINES	usbEhcdRhCreatePipe() – creates a pipe specific to an endpoint. usbEhcdRhDeletePipe() – deletes a pipe specific to an endpoint. usbEhcdRhSubmitURB() – submits a request to an endpoint. usbEhcdRhProcessControlRequest() – processes a control transfer request usbEhcdRhProcessInterruptRequest() – processes a interrupt transfer request usbEhcdRhProcessStandardRequest() – processes a standard transfer request

usbEhcdRhClearPortFeature() – clears a feature of the port
usbEhcdRhGetHubDescriptor() – get the hub descriptor
usbEhcdRhGetPortStatus() – get the status of the port
usbEhcdRhSetPortFeature() – set the features of the port
usbEhcdRhProcessClassSpecificRequest() – processes a class specific request
usbEhcdRHCeCancelURB() – cancels a request submitted for an endpoint

DESCRIPTION This contains functions which handle the requests to the Root hub

INCLUDE FILES **usb2/usbOsai.h** **usb2/usbHst.h** **usb2/usbEhcdDataStructures.h**,
usb2/usbEhcdRhEmulation.h **usb2/usbEhcdHal.h** **usb2/usbEhcdUtil.h**,
usb2/usbHcdInstr.h

usbEhcdTransferManagement

NAME **usbEhcdTransferManagement** – transfer management functions of the EHCD

ROUTINES

DESCRIPTION This module defines the interfaces which are registered with the USB D during EHCI Host Controller Driver initialization.

INCLUDE FILES **usbhst.h**, **usbEhcdDataStructures.h**, **usbEhcdInterfaces.h**, **usbEhcdUtil.h**,
usbEhcdConfig.h, **usbEhcdHal.h**, **usbEhcdRHEmulation.h**, **usbEhcdDebug.h**

SEE ALSO None

usbEhcdUtil

NAME **usbEhcdUtil** – contains the utility functions of EHCD

ROUTINES

DESCRIPTION This module defines the functions which serve as utility functions for the EHCI Host Controller Driver.

INCLUDE FILES **usbhst.h**, **usbEhcdDataStructures.h**, **usbEhcdUtil.h**, **usbEhcdDebug.h**

SEE ALSO *USB specification, revision 2.0, EHCI specification, revision 1.0*

usbHalDeviceControlStatus

NAME	usbHalDeviceControlStatus – HAL Device Control and Status handler module
ROUTINES	usbHalTcdAddressSet() – hal interface to set address. usbHalTcdSignalResume() – hal interface to initiate resume signal. usbHalTcdDeviceFeatureSet() – hal interface to set feature on the device. usbHalTcdDeviceFeatureClear() – hal interface to clear feature on device. usbHalTcdCurrentFrameGet() – hal interface to get Current Frame Number.
DESCRIPTION	This file contains device control and status handler routines of the Hardware Adaption Layer.
INCLUDE FILES	usb/target/usbHalLib.h usb/target/usbHal.h, usb/target/usbHalDebug.h, usb/target/usbPeriphInstr.h

usbHalEndpoint

NAME	usbHalEndpoint – HAL Endpoint specific functionalities
ROUTINES	usbHalTcdEndpointAssign() – configure an endpoint on the target controller usbHalTcdEndpointRelease() – unconfigure endpoint on the target controller usbHalTcdEndpointStateSet() – set the state of an endpoint usbHalTcdEndpointStatusGet() – get the status of an endpoint usbHalTcdErpSubmit() – submit an ERP for an endpoint usbHalTcdErpCancel() – cancel an ERP
DESCRIPTION	This file defines the hardware independent endpoint specific functionalities of the Hardware Adaption Layer.
INCLUDE FILES	drv/usb/target/usbTcd.h, usb/target/usbHal.h, usb/target/usbHalDebug.h, usb/ossLib.h, string.h, usb/target/usbPeriphInstr.h

usbHalInitExit

NAME	usbHalInitExit – HAL initialization and uninitialization functionalities
-------------	---------------------------------------------------------------------------------

ROUTINES	usbHalTcdAttach() – attaches a TCD usbHalTcdDetach() – detaches a TCD usbHalTcdEnable() – enables the target controller. usbHalTcdDisable() – disables the target controller
DESCRIPTION	This file defines the hardware independent initialization and uninitialization functions of the Hardware Adaption Layer.
INCLUDE FILES	drv/usb/target/usbTcd.h , usb/target/usbHal.h , usb/target/usbHalLib.h , usb/target/usbHalDebug.h , usb/ossLib.h , usb/target/usbPeriphInstr.h

usbHalInterruptHandler

NAME	usbHalInterruptHandler – USB HAL interrupt handler module
ROUTINES	
DESCRIPTION	This file contains interrupt handler routines of the Hardware Adaption Layer.
INCLUDE FILES	usb/target/usbHal.h , usb/target/usbHalDebug.h , usb/ossLib.h , usb/target/usbPeriphInstr.h

usbHalUtil

NAME	usbHalUtil – Utility functions of HAL
ROUTINES	
DESCRIPTION	This file defines the utility functions which are used by the sub-modules of the Hardware Adaption Layer.
INCLUDE FILES	usb/target/usbTcd.h , string.h

usbHandleLib

NAME	usbHandleLib – handle utility functions
------	------------------------------------------------

ROUTINES	usbHandleInitialize() – Initializes the handle utility library. usbHandleShutdown() – Shuts down the handle utility library. usbHandleCreate() – Creates a new handle. usbHandleDestroy() – Destroys a handle. usbHandleValidate() – Validates a handle.
DESCRIPTION	<p>Implements a set of general-purpose handle creation and validation functions.</p> <p>Using these services, libraries can return handles to callers which can subsequently be validated for authenticity. This provides libraries with an additional measure of bullet-proofing.</p>
INCLUDE FILES	usbHandleLib.h

usbHubInitialization

NAME	usbHubInitialization – Initialization and cleanup of HUB class driver
ROUTINES	usbHubInit() – registers USB Hub Class Driver function pointers. usbHubExit() – de-registers and cleans up the USB Hub Class Driver.
DESCRIPTION	This module provides the initialization and the clean up functions for the USB Hub Class Driver.
INCLUDE FILES	usb2/usbOsai.h, usb2/usbHubCommon.h, usb2/usbHubGlobalVariables.h, usb2/usbHubInitialization.h, usb2/usbHubClassInterface.h, usb2/usbHst.h, usb2/usbHcdInstr.h

usbKeyboardLib

NAME	usbKeyboardLib – USB keyboard class drive with vxWorks SIO interface
ROUTINES	usbKeyboardDevInit() – initialize USB keyboard SIO driver usbKeyboardDevShutdown() – shuts down keyboard SIO driver usbKeyboardDynamicAttachRegister() – Register keyboard attach callback usbKeyboardDynamicAttachUnregister() – Unregisters keyboard attach callback usbKeyboardSioChanLock() – Marks SIO_CHAN structure as in use usbKeyboardSioChanUnlock() – Marks SIO_CHAN structure as unused

DESCRIPTION

This module implements the USB keyboard class driver for the vxWorks operating system. This module presents an interface which is a superset of the vxWorks SIO (serial IO) driver model. That is, this driver presents the external APIs which would be expected of a standard "multi-mode serial (SIO) driver" and adds certain extensions which are needed to address adequately the requirements of the hot-plugging USB environment.

USB keyboards are described as part of the USB "human interface device" class specification and related documents. This driver concerns itself only with USB devices which claim to be keyboards as set forth in the USB HID specification and ignores other types of human interface devices (i.e., mouse). USB keyboards can operate according to either a "boot protocol" or to a "report protocol". This driver enables keyboards for operation using the boot protocol.

As the SIO driver model presents a fairly limited, byte-stream oriented view of a serial device, this driver maps USB keyboard scan codes into appropriate ASCII codes. Scan codes and combinations of scan codes which do not map to the ASCII character set are suppressed.

Unlike most SIO drivers, the number of channels supported by this driver is not fixed. Rather, USB keyboards may be added or removed from the system at any time. This creates a situation in which the number of channels is dynamic, and clients of **usbKeyboardLib.c** need to be made aware of the appearance and disappearance of channels. Therefore, this driver adds an additional set of functions which allows clients to register for notification upon the insertion and removal of USB keyboards, and hence the creation and deletion of channels.

This module itself is a client of the Universal Serial Bus Driver (USBBD). All interaction with the USB buses and devices is handled through the USBBD.

INITIALIZATION

As with standard SIO drivers, this driver must be initialized by calling **usbKeyboardDevInit()**. **usbKeyboardDevInit()** in turn initializes its connection to the USBBD and other internal resources needed for operation. Unlike some SIO drivers, there are no **usbKeyboardLib.c** data structures which need to be initialized prior to calling **usbKeyboardDevInit()**.

Prior to calling **usbKeyboardDevInit()**, the caller must ensure that the USBBD has been properly initialized by calling - at a minimum - **usbdInitialize()**. It is also the caller's responsibility to ensure that at least one USB HCD (USB Host Controller Driver) is attached to the USBBD - using the USBBD function **usbdHcdAttach()** - before keyboard operation can begin. However, it is not necessary for **usbdHcdAttach()** to be called prior to initializing **usbKeyboardLib.c**. **usbKeyboardLib.c** uses the USBBD dynamic attach services and is capable of recognizing USB keyboard attachment and removal on the fly. Therefore, it is possible for USB HCDs to be attached to or detached from the USBBD at run time - as may be required, for example, in systems supporting hot swapping of hardware.

usbKeyboardLib.c does not export entry points for transmit, receive, and error interrupt entry points like traditional SIO drivers. All "interrupt" driven behavior is managed by the underlying USBBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of **usbKeyboardLib.c**. For the same reason, there is no

post-interrupt-connect initialization code and **usbKeyboardLib.c** therefore also omits the "devInit2" entry point.

OTHER FUNCTIONS

usbKeyboardLib.c also supports the SIO ioctl interface. However, attempts to set parameters like baud rates and start/stop bits have no meaning in the USB environment and will be treated as no-ops.

DATA FLOW

For each USB keyboard connected to the system, **usbKeyboardLib.c** sets up a USB pipe to monitor input from the keyboard. Input, in the form of scan codes, is translated to ASCII codes and placed in an input queue. If SIO callbacks have been installed and **usbKeyboardLib.c** has been placed in the SIO "interrupt" mode of operation, then **usbKeyboardLib.c** will invoke the "character received" callback for each character in the queue. When **usbKeyboardLib.c** has been placed in the "polled" mode of operation, callbacks will not be invoked and the caller will be responsible for fetching keyboard input using the driver's **pollInput()** function.

usbKeyboardLib.c does not support output to the keyboard. Therefore, calls to the **txStartup()** and **pollOutput()** functions will fail. The only "output" supported is the control of the keyboard LEDs, and this is handled internally by **usbKeyboardLib.c**.

The caller needs to be aware that **usbKeyboardLib.c** is not capable of operating in a true "polled mode" as the underlying USB D and USB HCD always operate in an interrupt mode.

TYPEOMATIC REPEAT

USB keyboards do not implement typeomatic repeat, and it is the responsibility of the host software to implement this feature. For this purpose, this module creates a task called **typematicThread()** which monitors all open channels and injects repeated characters into input queues as appropriate.

INCLUDE FILES

sioLib.h, usbKeyboardLib.h

usbLib

NAME

usbLib – USB utility functions

ROUTINES

usbTransferTime() – Calculates the bus time required for a USB transfer.
usbRecurringTime() – calculates recurring time for interrupt/isoch transfers.
usbDescrParseSkip() – search for a descriptor and increment buffer.
usbDescrParse() – search a buffer for the a particular USB descriptor
usbConfigCountGet() – Retrieves the number of device configurations.
usbConfigDescrGet() – Reads the full configuration descriptor from device.
usbHidReportSet() – Issues a SET_REPORT request to a USB HID.

usbHidIdleSet() – Issues a **SET_IDLE** request to a USB HID.

usbHidProtocolSet() – Issues a **SET_PROTOCOL** request to a USB HID.

DESCRIPTION	This module contains miscellaneous functions which may be used by the USB driver (USBD), USB HCD (USB Host Controller Driver), or by USBD clients.
INCLUDE FILES	usbLib.h

usbListLib

NAME	usbListLib – Linked list utility functions
ROUTINES	usbListLink() – Adds an element to a linked list. usbListLinkProt() – Adds an element to a list guarded by a mutex. usbListUnlink() – Removes an entry from a linked list. usbListUnlinkProt() – Removes an element from a list guarded by a mutex. usbListFirst() – Returns first entry on a linked list. usbListNext() – Retrieves the next pStruct in a linked list.
DESCRIPTION	<p>This file implements a set of general-purpose linked-list functions which are portable across operating systems. Linked lists are collections of link structures. Each link structure contains forward and backward list pointers and a <i>pStruct</i> field which typically points to the caller's structure that contains the link structure.</p> <p>usbListLink() and usbListUnlink() are used to add and remove link structures in a linked list. The link field may be placed anywhere in the client's structure. The client's structure may even contain more than one link field, allowing the structure to be linked to multiple lists simultaneously.</p> <p>usbListFirst() retrieves the first structure on a linked list and usbListNext() retrieves subsequent structures.</p>
INCLUDE FILES	usbListLib.h

usbMouseLib

NAME	usbMouseLib – USB mouse class drive with vxWorks SIO interface
ROUTINES	usbMouseDevInit() – initialize USB mouse SIO driver usbMouseDevShutdown() – shuts down mouse SIO driver

usbMouseDynamicAttachRegister() – Register mouse attach callback
usbMouseDynamicAttachUnregister() – Unregisters mouse attach callback
usbMouseSioChanLock() – Marks SIO_CHAN structure as in use
usbMouseSioChanUnlock() – Marks SIO_CHAN structure as unused

DESCRIPTION

This module implements the USB mouse class driver for the vxWorks operating system. This module presents an interface which is a superset of the vxWorks SIO (serial IO) driver model. That is, this driver presents the external APIs which would be expected of a standard "multi-mode serial (SIO) driver" and adds certain extensions which are needed to address adequately the requirements of the hot-plugging USB environment.

USB mice are described as part of the USB "human interface device" class specification and related documents. This driver concerns itself only with USB devices which claim to be mice as set forth in the USB HID specification and ignores other types of human interface devices (i.e., keyboard). USB mice can operate according to either a "boot protocol" or to a "report protocol". This driver enables mice for operation using the boot protocol.

Unlike most SIO drivers, the number of channels supported by this driver is not fixed. Rather, USB mice may be added or removed from the system at any time. This creates a situation in which the number of channels is dynamic, and clients of **usbMouseLib.c** need to be made aware of the appearance and disappearance of channels. Therefore, this driver adds an additional set of functions which allows clients to register for notification upon the insertion and removal of USB mice, and hence the creation and deletion of channels.

This module itself is a client of the Universal Serial Bus Driver (USBBD). All interaction with the USB buses and devices is handled through the USBBD.

INITIALIZATION

As with standard SIO drivers, this driver must be initialized by calling **usbMouseDevInit()**. **usbMouseDevInit()** in turn initializes its connection to the USBBD and other internal resources needed for operation. Unlike some SIO drivers, there are no **usbMouseLib.c** data structures which need to be initialized prior to calling **usbMouseDevInit()**.

Prior to calling **usbMouseDevInit()**, the caller must ensure that the USBBD has been properly initialized by calling - at a minimum - **usbddInitialize()**. It is also the caller's responsibility to ensure that at least one USB HCD (USB Host Controller Driver) is attached to the USBBD - using the USBBD function **usbddHcdAttach()** - before mouse operation can begin. However, it is not necessary for **usbddHcdAttach()** to be called prior to initializing **usbMouseLib.c**. **usbMouseLib.c** uses the USBBD dynamic attach services and is capable of recognizing USB keyboard attachment and removal on the fly. Therefore, it is possible for USB HCDs to be attached to or detached from the USBBD at run time - as may be required, for example, in systems supporting hot swapping of hardware.

usbMouseLib.c does not export entry points for transmit, receive, and error interrupt entry points like traditional SIO drivers. All "interrupt" driven behavior is managed by the underlying USBBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of **usbMouseLib.c**. For the same reason, there is no

post-interrupt-connect initialization code and **usbKeyboardLib.c** therefore also omits the "devInit2" entry point.

OTHER FUNCTIONS

usbMouseLib.c also supports the SIO ioctl interface. However, attempts to set parameters like baud rates and start/stop bits have no meaning in the USB environment and will be treated as no-ops.

DATA FLOW

For each USB mouse connected to the system, **usbMouseLib.c** sets up a USB pipe to monitor input from the mouse. **usbMouseLib.c** supports only the SIO "interrupt" mode of operation. In this mode, the application must install a "report callback" through the driver's **callbackInstall()** function. This callback is of the form:

```
typedef STATUS (*REPORT_CALLBACK)
(
    void *arg,
    pHID_MSE_BOOT_REPORT pReport
);
```

usbMouseLib.c will invoke this callback for each report received. The STATUS returned by the callback is ignored by **usbMouseLib.c**. If the application is unable to accept a report, the report is discarded. The report structure is defined in **usbHid.h**, which is included automatically by **usbMouseLib.h**.

usbMouseLib.c does not support output to the mouse. Therefore, calls to the **txStartup()** and **pollOutput()** functions will fail.

INCLUDE FILES

sioLib.h, **usbMouseLib.h**

usbOhci

NAME

usbOhci – USB OHCI Driver Entry and Exit points

ROUTINES

usbOhciInstantiate() – instantiate the USB OHCI Host Controller Driver.
usbOhciInit() – initialize the USB OHCI Host Controller Driver.
usbOhciExit() – uninitialize the USB OHCI Host Controller Driver.
vxUsbOhciRegister() – registers OHCI driver with vxBus

DESCRIPTION

This provides the entry and exit points for the USB OHCI driver.

INCLUDE FILES

usbOhci.h, **usbOhciRegisterInfo.h**, **usbOhciTransferManagement.h**,
usbOhciRootHubEmulation.c, **usbOhciTransferManagement.c**, **usbOhciIsr.c**,
rebootLib.h

usbOhciDebug

NAME	usbOhciDebug – USB OHCI Debug Routines
ROUTINES	usbOhciDumpRegisters() – dump registers contents. usbOhciDumpMemory() – dump memory contents usbOhciDumpEndpointDescriptor() – dump endpoint descriptor contents usbOhciDumpPeriodicEndpointList() – dump periodic endpoint descriptor list usbOhciDumpGeneralTransferDescriptor() – dump general transfer descriptor usbOhciDumpPendingTransfers() – dump pending transfers usbOhciInitializeModuleTestingFunctions() – obtains entry points
DESCRIPTION	This file contains functions for display the USB OHCI registers, memory, endpoint descriptor, transfer descriptor etc. This interfaces exposed from this file can be used to debug the OHCI driver.
INCLUDE FILES	usbOhci.h, usbOhciRegisterInfo.h, usbOhciTransferManagement.h

usbPegasusEnd

NAME	usbPegasusEnd – USB Ethernet driver for the Pegasus USB-Ethernet adapter
ROUTINES	usbPegasusEndInit() – initializes the pegasus library pegasusMuxTxRestart() – place muxTxRestart on netJobRing pegasusOutIrpInUse() – determines if any of the output IRP's are in use usbPegasusEndLoad() – initialize the driver and device usbPegasusDynamicAttachRegister() – register PEGASUS device attach callback usbPegasusDynamicAttachUnregister() – unregisters PEGASUS attach callback usbPegasusDevLock() – marks USB_PEGASUS_DEV structure as in use usbPegasusDevUnlock() – marks USB_PEGASUS_DEV structure as unused usbPegasusReadReg() – read contents of specified and print usbPegasusEndUninit() – un-initializes the pegasus class driver
DESCRIPTION	This module is the USB communication class, Ethernet Sub class driver for the vxWorks operating system. This module presents an interface which becomes an underlying layer of the vxWorks END (Enhanced Network Driver) model. It also adds certain APIs that are necessary for some additional features supported by an usb - Ethernet adapter.

USB - Ethernet adapter devices are described in the USB Communication Devices class definitions. The USB - Ethernet adapter falls under the Ethernet Control model under the communications device class specification. This driver is meant for the usb-ethernet adapters built around the Pegasus-ADM Tek AN986 chip.

DEVICE FUNCTIONALITY

The Pegasus USB to ethernet adapter chip ASIC provides bridge from USB to 10/100 MII and USB to 1M HomePNA network. The Pegasus Chip, is compliant with supports USB 1.0 and 1.1 specifications. This device supports 4 End Points. The first, is the default end point which is of control type (with max 8 byte packet). The Second and the Third are BULK IN (Max 64 Byte packet) and BULK OUT (Max 64 Byte Packet) end points for transferring the data into the Host and from the Host respectively. The Fourth End Point, is an Interrupt end point (Max 8 bytes) that is not currently used.

This device supports One configuration which contains One Interface. This interface contains the 3 end points i.e. the Bulk IN/Out and interrupt end points.

Apart from the traditional commands, the device supports 3 Vendor specific commands. These commands are described in the Pegasus specification manual. The device supports interface to EEPROM for storing the Ethernet MAC address and other configuration details. It also supports interface to SRAM for storing the packets received and to be transmitted.

Packets are passed between the chip and host via bulk transfers. There is an interrupt endpoint mentioned in the specification manual. However it was not used. This device can work in 10Mbps half and Full duplex and 100 Mbps half and Full Duplex modes. The MAC supports a 64 entry multicast filter. This device is IEEE 802.3 MII compliant and supports IEEE 802.3x flow control. It also supports for configurable threshold for transmitting PAUSE frame. Supports Wakeup frame, Link status change and magic packet frame.

The device supports the following (vendor specific) commands :

USB_REQ_REG_GET

Retrieves the Contents of the specified register from the device.

USB_REQ_REG_SET_SINGLE

Sets the contents of the specified register (Single) in the device

USB_REQ_REG_SET_MULTIPLE

Sets the contents of the specified register (Multiple) in the device

DRIVER FUNCTIONALITY

The function **usbPegasusEndInit()** is called at the time of usb system initialization. It registers as a client with the USB D. This function also registers for the dynamic attachment and removal of the usb devices. Ideally we should be registering for a specific Class ID and a Subclass Id..but since the device doesn't support these parameters in the Device descriptor, we register for ALL kinds of devices. We maintain a linked list of the ethernet devices on USB in a linked list "pegasusDevList". This list is created and maintained using the linked list library provided as a part of the USB D. Useful API calls are provided to find if the device exists in the list, by taking either the device "nodeId" or the vendorId and

productId as the parameters. The Callback function registered for the dynamic attachment/removal, **pegasusAttachCallback()** will be called if any device is found on/removed from the USB. This function first checks whether the device already exists in the List. If not, it will parse through the device descriptor, find out the Vendor Id and Product Id. If they match with Pegasus Ids, the device will be added to the list of ethernet devices found on the USB.

pegasusDevInit() does most of the device structure initialization afterwards. This routine checks if the device corresponding to the `nodeId` matches to any of the devices in the `pegasusDevList`. If yes a pointer structure on the list will be assigned to one of the device structure parameters. After this the driver will parse through the configuration descriptor, interface descriptor to find out the InPut and OutPut end point details. Once we find these end point descriptors we create input and output Pipes and assign them to the corresponding structure. It then resets the device.

This driver, is a Polled mode driver as such. It keeps listening on the input pipe by calling "pegasusListenToInput" all the time, from the first time it is called by **pegasusStart()**. This acquires a buffer from the `endLayer` and uses it in the IRP. Unless the IRP is cancelled (by **pegasusStop()**), it will be submitted again and again. If cancelled, it will again start listening only if **pegasusStart()** is called. If there is data (IRP successfull), then it will be passed on to END by calling **pegasusEndRecv()**.

Rest of the functionality of the driver is straight forward and most of the places achieved by sending a vendor specific command from the list described above, to the device.

INCLUDE FILES `end.h, endLib.h, lstLib.h, etherMultiLib.h, usb/usbPlatform.h, usb/usb.h, usb/usbListLib.h, usb/usbdLib.h, usb/usbLib.h, drv/usb/usbPegasusEnd.h`

SEE ALSO `muxLib, endLib, usbLib, usbdLib, ossLib, "Writing and Enhanced Network Driver" and, "USB Developer's Kit User's Guide"`

usbPrinterLib

NAME `usbPrinterLib` – USB printer class drive with vxWorks SIO interface

ROUTINES

- `usbPrinterDevInit()` – initialize USB printer SIO driver
- `usbPrinterDevShutdown()` – shuts down printer SIO driver
- `usbPrinterDynamicAttachRegister()` – Register printer attach callback
- `usbPrinterDynamicAttachUnregister()` – Unregisters printer attach callback
- `usbPrinterSioChanLock()` – Marks `SIO_CHAN` structure as in use
- `usbPrinterSioChanUnlock()` – Marks `SIO_CHAN` structure as unused

DESCRIPTION This module implements the USB printer class driver for the vxWorks operating system. This module presents an interface which is a superset of the vxWorks SIO (serial IO) driver

model. That is, this driver presents the external APIs which would be expected of a standard "multi-mode serial (SIO) driver" and adds certain extensions which are needed to address adequately the requirements of the hot-plugging USB environment.

USB printers are described in the USB Printer Class definition. This class driver specification presents two kinds of printer: uni-directional printers (output only) and bi-directional printers (capable of both output and input). This class driver is capable of handling both kinds of printers. If a printer is uni-directional, then the SIO driver interface only allows characters to be written to the printer. If the printer is bi-directional, then the SIO interface allows both output and input streams to be written/read.

Unlike most SIO drivers, the number of channels supported by this driver is not fixed. Rather, USB printers may be added or removed from the system at any time. This creates a situation in which the number of channels is dynamic, and clients of **usbPrinterLib.c** need to be made aware of the appearance and disappearance of channels. Therefore, this driver adds an additional set of functions which allows clients to register for notification upon the insertion and removal of USB printers, and hence the creation and deletion of channels.

This module itself is a client of the Universal Serial Bus Driver (USBBD). All interaction with the USB buses and devices is handled through the USBBD.

INITIALIZATION

As with standard SIO drivers, this driver must be initialized by calling **usbPrinterDevInit()**. **usbPrinterDevInit()** in turn initializes its connection to the USBBD and other internal resources needed for operation. Unlike some SIO drivers, there are no **usbPrinterLib.c** data structures which need to be initialized prior to calling **usbPrinterDevInit()**.

Prior to calling **usbPrinterDevInit()**, the caller must ensure that the USBBD has been properly initialized by calling - at a minimum - **usbddInitialize()**. It is also the caller's responsibility to ensure that at least one USB HCD (USB Host Controller Driver) is attached to the USBBD - using the USBBD function **usbddHcdAttach()** - before printer operation can begin. However, it is not necessary for **usbddHcdAttach()** to be called prior to initializing **usbPrinterLib.c**. **usbPrinterLib.c** uses the USBBD dynamic attach services and is capable of recognizing USB printer attachment and removal on the fly. Therefore, it is possible for USB HCDs to be attached to or detached from the USBBD at run time - as may be required, for example, in systems supporting hot swapping of hardware.

usbPrinterLib.c does not export entry points for transmit, receive, and error interrupt entry points like traditional SIO drivers. All "interrupt" driven behavior is managed by the underlying USBBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of **usbPrinterLib.c**. For the same reason, there is no post-interrupt-connect initialization code and **usbPrinterLib.c** therefore also omits the "devInit2" entry point.

OTHER FUNCTIONS

usbPrinterLib.c also supports the SIO ioctl interface. However, attempts to set parameters like baud rates and start/stop bits have no meaning in the USB environment and will be treated as no-ops.

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

DATA FLOW For each USB printer connected to the system, **usbPrinterLib.c** sets up a USB pipe to output bulk data to the printer. This is the pipe through which printer control and page description data will be sent to the printer. Additionally, if the printer is bi-directional, **usbPrinterLib.c** also sets up a USB pipe to receive bulk input data from the printer. The meaning of data received from a bi-directional printer depends on the specific make/model of printer.

The USB printer SIO driver supports only the SIO "interrupt" mode of operation - **SIO_MODE_INT**. Any attempt to place the driver in the polled mode will return an error.

INCLUDE FILES **sioLib.h, usbPrinterLib.h**

usbQueueLib

NAME **usbQueueLib** – O/S-independent queue functions

ROUTINES **usbQueueCreate()** – Creates an OS-independent queue structure.
usbQueueDestroy() – Destroys a queue.
usbQueuePut() – Puts a message into a queue.
usbQueueGet() – Retrieves a message from a queue.

DESCRIPTION This file contains a generic implementation of operating system-independent queue routines which are built on top of the the **ossLib** library's mutex and semaphore routines.

The caller creates a queue of depth "n" by calling **usbQueueCreate()** and receives a **QUEUE_HANDLE** in response. The **QUEUE_HANDLE** must be used in all subsequent calls to **usbQueuePut()**, **usbQueueGet()**, and **usbQueueDestroy()**.

Each entry in a queue is described by a **USB_QUEUE** structure which contains *msg*, *wParam*, and *lParam* fields. The values of these fields are arbitrary and may be used in any way by the calling application.

INCLUDE FILES **usbQueueLib.h**

usbSpeakerLib

NAME	usbSpeakerLib – USB speaker class drive with vxWorks SEQ_DEV interface
ROUTINES	usbSpeakerDevInit() – initialize USB speaker SIO driver usbSpeakerDevShutdown() – shuts down speaker SIO driver usbSpeakerDynamicAttachRegister() – Register speaker attach callback usbSpeakerDynamicAttachUnregister() – Unregisters speaker attach callback usbSpeakerSeqDevLock() – Marks SEQ_DEV structure as in use usbSpeakerSeqDevUnlock() – Marks SEQ_DEV structure as unused
DESCRIPTION	<p>This module implements the class driver for USB audio devices. USB audio devices are a subset of the USB audio class, and this module handles only those parts of the USB audio class definition which are relevant to the operation of USB speakers and microphones.</p> <p>This module presents a modified VxWorks SEQ_DEV interface to its callers. The SEQ_DEV interface was chosen because, of the existing VxWorks driver models, it 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 the USB and to audio devices in general. Functions have been added to allow callers to recognize the dynamic attachment and removal of speaker devices. IOCTL functions have been added to retrieve and control additional settings related to speaker operation.</p> <p>This usbSpeakerLib has been enhanced from previously releases to support USB microphones usually in the form of USB headsets.</p>
INITIALIZATION	<p>As with standard SEQ_DEV drivers, this driver must be initialized by calling usbSpeakerDevInit(). usbSpeakerDevInit() in turn initializes its connection to the USBDB and other internal resources needed for operation. Unlike some SEQ_DEV drivers, there are no usbSpeakerLib.c data structures which need to be initialized prior to calling usbSpeakerDevInit().</p> <p>Prior to calling usbSpeakerDevInit(), the caller must ensure that the USBDB has been properly initialized by calling - at a minimum - usbdbInitialize(). It is also the caller's responsibility to ensure that at least one USB HCD (USB Host Controller Driver) is attached to the USBDB.</p> <p>usbSpeakerLib.c uses the USBDB dynamic attach services and is capable of recognizing USB audio devices attachment and removal on the fly. Therefore, it is possible for USB HCDs to be attached to or detached from the USBDB at run time - as may be required, for example, in systems supporting hot swapping of hardware.</p> <p>RECOGNIZING & HANDLING USB SPEAKERS</p> <p>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</p>

"Output Terminal". For each USB audio device, **usbSpeakerLib** examines the descriptors which enumerate the "units" and "terminals" contained within the device. These descriptors define both which kinds of units/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 like volume. Once having built such an internal "map" of the device, **usbSpeakerLib** configures the device and waits for a caller to provide a stream of audio data. If no "Output Terminal" is found, **usbSpeakerLib** ignores the audio device.

After determining that the audio device contains an Output Terminal, **usbSpeakerLib** builds a list of the audio formats supported by the device. **usbSpeakerLib** supports only AudioStreaming interfaces (no MidiStreaming is supported).

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

DYNAMIC ATTACHMENT & REMOVAL OF SPEAKERS

As with other USB devices, USB speakers may be attached to or detached from the system dynamically. **usbSpeakerLib** uses the USBBD's dynamic attach services 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 indicated 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 **usbSpeakSeqDevLock()** and can decrement the usage count by calling **usbSpeakerSeqDevUnlock()**. When a USB audio device is removed from the system and its usage count is 0, **usbSpeakerLib** automatically removes all data structures, including the **SEQ_DEV** structure itself, 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/unlock functions provide a mechanism for callers to protect these data structures as needed.

RECOGNIZING & HANDLING USB MICROPHONES

As with other USB speakers, microphones may be attached to or detached from the system dynamically. **usbSpeakerLib** uses the USBBD'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 indicated whether the microphone is being attached or removed.

DATA FLOW - Speakers

Before sending audio data to a speaker device, the caller must specify the data format (e.g., PCM, MPEG) using an IOCTL (see below). The USB speaker itself must support the indicated (or a similar) data format.

USB speakers rely on an uninterrupted, time-critical stream of audio data. The data is sent to the speaker through an isochronous pipe. In order for the data flow to continue uninterrupted, **usbSpeakerLib** internally uses a double-buffering scheme. When data is presented to **usbSpeakerLib**'s **sd_seqWrt()** function by the caller, **usbSpeakerLib** copies the data into an internal buffer and immediately releases the caller's buffer. The caller should immediately try to pass the next buffer to **usbSpeakerLib**. When **usbSpeakerLib**'s internal buffer is filled, it will block the caller until such time as it can accept the new data. In this manner, the caller and **usbSpeakerLib** work together to ensure that an adequate supply of audio data will always be available to continue isochronous transmission uninterrupted.

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 IOCTLs to "open" and "close" each audio stream. **usbSpeakerLib** relies on these IOCTLs to manage its internal buffers correctly.

DATA FLOW - Microphone

When connecting a microphone, the caller must select from the formats available on the microphone, a format appropriate for the desired application. Then using IOCTLs, the caller specifies the format and the interval in which the caller will obtain the data from the **usbSpeakerLib**. The **usbSpeakerLib** will then allocate an appropriate buffer size and post isochronous IN requests to the USB microphone, filling the buffer as the IN requests complete.

The caller then posts a **sd_seqRd** to the **usbSpeakerLib**, at appropriate intervals, to obtain the audio for further processing. Note that the reader will not block waiting for data, but will return all available data up to the requested buffer size. The caller should always check the return value of the read.

As with the USB speakers, double buffering is used to attempt to provide a continuous stream of data, however if caller cannot service the data at a sustainable rate, overwriting of the data buffers may occur.

A demonstration program using a USB headset (both speaker and microphone) is provided in source form as a configlet. See the Headset Denonstration section in the documentation

INCLUDE FILES **seqIo.h, usbAudio.h, usbSpeakerLib.h**

usbTargDefaultPipe

NAME **usbTargDefaultPipe** – Handles the requests to the default control pipe

ROUTINES	usbTargControlResponseSend() – sends data to host on the control pipe usbTargControlStatusSend() – sends control transfer status to the host usbTargControlPayloadRcv() – receives data on the default control pipe usbTargSetupErpCallback() – handles the setup packet
DESCRIPTION	This module handles the standard requests to the default pipe by calling the callback functions present in the callback table. It also provides the interfaces for non-standard control data transfers on the default control pipe to the USB Target Application.
INCLUDE FILES	usb/usbPlatform.h, string.h, usb/ossLib.h, usb/usb.h, usb/usbHandleLib.h, usb/target/HalLib.h, usb/target/usbHalCommon.h, usb/target/usbTargLib.h, usb/target/usbTargUtil.h, usb/target/usbPeriphInstr.h

usbTargDeviceControl

NAME	usbTargDeviceControl – modules for handling pipe specific requests
ROUTINES	usbTargCurrentFrameGet() – retrieves the current USB frame number usbTargSignalResume() – drives RESUME signalling on USB usbTargDeviceFeatureSet() – sets or enable a specific feature usbTargDeviceFeatureClear() – clears a specific feature usbTargMgmtCallback() – invoked when HAL detects a management event
DESCRIPTION	This module provides interfaces for handling device control and status requests.
INCLUDE FILES	usb/usbPlatform.h, string.h, usb/ossLib.h, usb/usb.h, usb/usbHandleLib.h, usb/target/HalLib.h, usb/target/usbHalCommon.h, usb/target/usbTargLib.h, usb/target/usbTargUtil.h, usb/target/usbPeriphInstr.h

usbTargInitExit

NAME	usbTargInitExit – USB Initialization/Uninitialization modules
ROUTINES	usbTargInitialize() – initializes the USB Target Library usbTargShutdown() – shutdown the USB target library usbTargTcdAttach() – to attach the TCD to the target library usbTargTcdDetach() – detaches a USB target controller driver usbTargEnable() – enables target channel onto USB usbTargDisable() – disables a target channel

DESCRIPTION	<p>This module implements the hardware-independent USB target API. It provides the required interfaces for initializing and un-initializing the USB Target Library and the TCD.</p> <p>USB Target Library must be initialized by calling usbTargInitialize(). Before operation can begin, at least one TCD must be attached to usb Target Library by calling usbTargTcdAttach(). In response to a successful TCD attachment. A handle is returned. This handle must be used in all subsequent calls to usbTargLib to identify a given target channel.</p> <p>USB devices (targets) almost never initiate activity on the USB (the exception being RESUME signalling). So, as part of the call to usbTargTcdAttach(), the caller must provide a pointer to a USB_TARG_CALLBACK_TABLE structure. This table contains a collection of callback function pointers initialized by the caller prior to invoking the usbTargTcdAttach() function. Through these callbacks, usbTargLib notifies the calling application of various USB events and requests from the host.</p>
INCLUDE FILES	usb/usbPlatform.h, usb/ossLib.h, usb/usb.h, usb/usbListLib.h, usb/usbHandleLib.h, usb/target/HalLib.h, usb/target/usbHalCommon.h, usb/target/usbTargLib.h, usb/target/usbTargUtil.h, usb/target/usbPeriphInstr.h

usbTargKbdLib

NAME	usbTargKbdLib – USB keyboard target exerciser/demonstration
ROUTINES	usbTargKbdCallbackInfo() – returns usbTargKbdLib callback table usbTargKbdInjectReport() – injects a "boot report"
DESCRIPTION	<p>This module contains code to exercise the usbTargLib by emulating a rudimentary USB keyboard. This module will generally be invoked by usbTool or a similar USB test/exerciser application.</p> <p>It is the caller's responsibility to initialize usbTargLib and attach a USB TCD to it. When attaching a TCD to usbTargLib, the caller must pass a pointer to a table of callbacks required by usbTargLib. The address of this table and the "callback parameter" required by these callbacks may be obtained by calling usbTargKbdCallbackInfo(). It is not necessary to initialize the usbTargKbdLib or to shut it down. It performs all of its operations in response to callbacks from usbTargLib.</p> <p>This module also exports a function called usbTargKbdInjectReport(). This function allows the caller to inject a "boot report" into the interrupt pipe. This allows for rudimentary emulation of keystrokes.</p>
INCLUDE FILES	usb/usbPlatform.h, string.h, usb/usb.h, usb/usbHid.h, usb/usbDescrCopyLib.h, usb/target/usbTargLib.h, drv/usb/target/usbTargKbdLib.h

usbTargMsLib

NAME	usbTargMsLib – Mass Storage routine library
ROUTINES	<p>usbMsCBWGet() – get the last mass storage CBW received</p> <p>usbMsCBWInit() – initialize the mass storage CBW</p> <p>usbMsCSWGet() – get the current CSW</p> <p>usbMsCSWInit() – initialize the CSW</p> <p>usbMsBulkInStall() – stall the bulk-in pipe</p> <p>usbMsBulkInUnStall() – unSTALL the bulk-in pipe</p> <p>usbMsBulkOutStall() – stall the bulk-out pipe</p> <p>usbMsBulkOutUnStall() – unSTALL the bulk-out pipe</p> <p>usbTargMsCallbackInfo() – returns usbTargPrnLib callback table</p> <p>usbMsBulkInErpInit() – initialize the bulk-in ERP</p> <p>usbMsBulkOutErpInit() – initialize the bulk-Out ERP</p> <p>usbMsIsConfigured() – test if the device is configured</p> <p>usbMsBulkInErpInUseFlagGet() – get the Bulk-in ERP inuse flag</p> <p>usbMsBulkOutErpInUseFlagGet() – get the Bulk-Out ERP inuse flag</p> <p>usbMsBulkInErpInUseFlagSet() – set the Bulk-In ERP inuse flag</p> <p>usbMsBulkOutErpInUseFlagSet() – set the Bulk-Out ERP inuse flag</p> <p>usbMsTestTxCallback() – invoked after test data transmitted</p> <p>usbMsTestRxCallback() – invoked after test data is received</p>
DESCRIPTION	This module defines those routines directly referenced by the USB peripheral stack; namely, the routines that initialize the USB_TARG_CALLBACK_TABLE data structure. Additional routines are also provided which are specific to the mass storage driver.
INCLUDES	<p>vxWorks.h, stdio.h, errnoLib.h, logLib.h, string.h, blkIo.h, usb/usbPlatform.h, usb/usb.h,</p> <p>usb/usbDescrCopyLib.h, usb/usbLib.h, usb/target/usbTargLib.h,</p> <p>drv/usb/usbBulkDevLib.h, drv/usb/target/usbTargMsLib.h,</p> <p>drv/usb/target/usbTargRbcLib.h</p>

usbTargPipeFunc

NAME	usbTargPipeFunc – modules for handling pipe specific requests
ROUTINES	<p>usbTargPipeCreate() – creates a pipe for communication on an endpoint</p> <p>usbTargPipeDestroy() – destroys an endpoint pipe</p> <p>usbTargTransfer() – to transfer data through a pipe</p> <p>usbTargTransferAbort() – cancels a previously submitted USB_ERP</p> <p>usbTargPipeStatusSet() – sets pipe stalled/unstalled status</p>

usbTargPipeStatusGet() – returns the endpoint status

DESCRIPTION	<p>This module provides interfaces for handling the various pipe specific requests.</p> <p>It provides interfaces for creating and destroying pipes, submit and cancel ERPs and to get and set the pipe status information.</p>
INCLUDE FILES	<p>usb/usbPlatform.h, string.h, usb/ossLib.h, usb/usb.h, usb/usbHandleLib.h, usb/target/HalLib.h, usb/target/usbHalCommon.h, usb/target/usbTargLib.h, usb/target/usbTargUtil.h, usb/target/usbPeriphInstr.h</p>

usbTargPrnLib

NAME	<p>usbTargPrnLib – USB printer target exerciser/demonstration</p>
ROUTINES	<p>usbTargPrnCallbackInfo() – returns usbTargPrnLib callback table</p> <p>usbTargPrnDataInfo() – returns buffer status/info</p> <p>usbTargPrnDataRestart() – restarts listening ERP</p>
DESCRIPTION	<p>This module contains code to exercise the usbTargLib by emulating a rudimentary USB printer. This module will generally be invoked by usbTool or a similar USB test/exerciser application.</p> <p>It is the caller's responsibility to initialize usbTargLib and attach a USB TCD to it. When attaching a TCD to usbTargLib, the caller must pass a pointer to a table of callbacks required by usbTargLib. The address of this table and the "callback parameter" required by these callbacks may be obtained by calling usbTargPrnCallbackInfo(). It is not necessary to initialize the usbTargPrnLib or to shut it down. It performs all of its operations in response to callbacks from usbTargLib.</p> <p>This module also exports a function, usbTargPrnBfrInfo(), which allows a test application to retrieve the current status of the bulk output buffer.</p>
INCLUDE FILES	<p>usb/usbPlatform.h, string.h, usb/usb.h, usb/usbPrinter.h, usb/usbDescrCopyLib.h, usb/target/usbTargLib.h, drv/usb/target/usbTargPrnLib.h, usb/target/usbHalCommon.h</p>

usbTargRbcCmd

NAME	<p>usbTargRbcCmd – Reduced Block Command set routine library</p>
-------------	-------------------------------------------------------------------------

ROUTINES	<p> usbTargRbcRead() – read data from the RBC device usbTargRbcCapacityRead() – read the capacity of the RBC device usbTargRbcStartStop() – start or stop the RBC device usbTargRbcPreventAllowRemoval() – prevent or allow the removal of the RBC device usbTargRbcVerify() – verify the last data written to the RBC device usbTargRbcWrite() – write to the RBC device usbTargRbcInquiry() – retrieve inquiry data from the RBC device usbTargRbcModeSelect() – select the mode parameter page of the RBC device usbTargRbcModeSense() – retrieve sense data from the RBC device usbTargRbcModeSelect10() – select the mode parameter page of the RBC device usbTargRbcModeSense10() – request for mode sense 10 command usbTargRbcTestUnitReady() – test if the RBC device is ready usbTargRbcBufferWrite() – write micro-code to the RBC device usbTargRbcFormat() – format the RBC device usbTargRbcPersistentReserveIn() – send reserve data to the host usbTargRbcPersistentReserveOut() – reserve resources on the RBC device usbTargRbcRelease() – release a resource on the RBC device usbTargRbcRequestSense() – request sense data from the RBC device usbTargRbcReserve() – reserve a resource on the RBC device usbTargRbcCacheSync() – synchronize the cache of the RBC device usbTargRbcBlockDevGet() – return opaque pointer to the RBC BLK I/O DEV device usbTargRbcBlockDevSet() – set the pointer to the RBC BLK I/O DEV device structure. usbTargRbcBlockDevCreate() – create an RBC BLK_DEV device. usbTargRbcVendorSpecific() – vendor specific call </p>
DESCRIPTION	<p>This module implements a framework based on the RBC (Reduced Block Command) set. These routines are invoked by the USB 2.0 mass storage driver based on the contents of the USB CBW (command block wrapper).</p>
INCLUDES	<p> vxWorks.h, disFsLib.h, dcacheCbio.h, ramDrv.h, usrFdiskPartLib.h, usb/usbPlatform.h, usb/usb.h, usb/target/usbTargLib.h, drv/usb/target/usbTargMsLib.h, drv/usb/target/usbTargRbcCmd.h, drv/xbd/xbd.h, xbdRamDisk.h </p>

usbTargRbcLib

NAME	usbTargRbcLib – USB Reduced Block Command set routine library
ROUTINES	<p> bulkOutErpCallbackCBW() – process the CBW on bulk-out pipe bulkInErpCallbackCSW() – send the CSW on bulk-in pipe bulkInErpCallbackData() – process end of data phase on bulk-in pipe bulkOutErpCallbackData() – process end of data phase on bulk-out pipe </p>

DESCRIPTION	This module defines the USB_ERP callback routines directly used by the USB 2.0 mass storage driver. These callback routines invoke the routines defined in the file usbTargRbcCmd.c .
INCLUDES	vxWorks.h , ramDrv.h , cbioLib.h , logLib.h , usb/usbPlatform.h , usb/usb.h , usb/usbdLib.h , usb/target/usbTargLib.h , drv/usb/usbBulkDevLib.h , drv/usb/target/usbTargMsLib.h , drv/usb/target/usbTargRbcCmd.h , drv/xbd/xbd.h

usbTargUtil

NAME	usbTargUtil – Utility Functions
ROUTINES	
DESCRIPTION	This file consists of utility functions which are used by the usbTarget Library files.
INCLUDE FILES	usb/usbPlatform.h , string.h , usb/ossLib.h , usb/usb.h , usb/usbHandleLib.h , usb/target/HalLib.h , usb/target/usbTargLib.h , usb/target/usbTargUtil.h

usbTcdIsp1582InitExit

NAME	usbTcdIsp1582InitExit – Initialization/uninitialization for ISP 1582 TCD
ROUTINES	usbTcdIsp1582EvalExec() – single Entry Point for ISP 1582 TCD
DESCRIPTION	<p>This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Philips ISP 1582.</p> <p>This module exports a single entry point, usbTcdIsp1582EvalExec(). This is the USB_TCD_EXEC_FUNC for this TCD. The caller passes requests to the TCD by constructing TRBs, or Target Request Blocks, and passing them to this entry point.</p> <p>TCDs are initialized by invoking the TCD_FNC_ATTACH function. In response to this function, the TCD returns information about the target controller, including its USB speed, the number of endpoints it supports etc.</p>
INCLUDE FILES	usb/usbPlatform.h , usb/ossLib.h , usb/usbPciLib.h , usb/target/usbHalCommon.h , usb/target/usbTcd.h , drv/usb/target/usbIsp1582Eval.h , drv/usb/target/usbTcdIsp1582EvalLib.h , drv/usb/target/usbIsp1582Tcd.h , drv/usb/target/usbIsp1582Debug.h , rebootLib.h , usb/target/usbPeriphInstr.h

usbTcdNET2280InitExit

NAME	usbTcdNET2280InitExit – initialization/uninitialization for NET2280 TCD
ROUTINES	usbTcdNET2280Exec() – single Entry Point for NETCHIP 2280 TCD
DESCRIPTION	<p>This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Netchip NET2280.</p> <p>This module exports a single entry point, usbTcdNET2280Exec(). This is the USB_TCD_EXEC_FUNC for this TCD. The caller passes requests to the TCD by constructing TRBs, or Target Request Blocks, and passing them to this entry point.</p> <p>TCDs are initialized by invoking the TCD_FNC_ATTACH function. In response to this function, the TCD returns information about the target controller, including its USB speed, the number of endpoints it supports etc.</p>
INCLUDE FILES	usb/usbPlatform.h, usb/ossLib.h, usb/usbPciLib.h, usb/target/usbHalCommon.h, usb/target/usbTcd.h, drv/usb/target/usbNET2280.h, drv/usb/target/usbNET2280Tcd.h, drv/usb/target/usbTcdNET2280Lib.h, drv/usb/target/usbTcdNET2280Debug.h, rebootLib.h, usb/target/usbPeriphInstr.h

usbTcdPdiusbd12InitExit

NAME	usbTcdPdiusbd12InitExit – Initialization/uninitialization for PDIUSB12 TCD
ROUTINES	usbTcdPdiusbd12EvalExec() – single entry point for PDIUSB12 TCD
DESCRIPTION	<p>This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Philips PDIUSB12.</p> <p>This module exports a single entry point, usbTcdPdiusbd12EvalExec(). This is the USB_TCD_EXEC_FUNC for this TCD. The caller passes requests to the TCD by constructing TRBs, or Target Request Blocks, and passing them to this entry point.</p> <p>TCDs are initialized by invoking the TCD_FNC_ATTACH function. In response to this function, the TCD returns information about the target controller, including its USB speed, the number of endpoints it supports etc.</p>
INCLUDE FILES	usb/usbPlatform.h, usb/ossLib.h, usb/target/usbIsaLib.h, drv/usb/target/usbPdiusbd12Eval.h, drv/usb/target/usbTcdPdiusbd12EvalLib.h, drv/usb/target/usbPdiusbd12Tcd.h, drv/usb/target/usbPdiusbd12Debug.h, usb/target/usbPeriphInstr.h

usbTransUnitData

NAME	usbTransUnitData – Translation Unit Data Transfer Interfaces
ROUTINES	usbPipeCreate() – Creates a USB pipe for subsequent transfers. usbPipeDestroy() – Destroys a USB data transfer pipe. usbTransfer() – Initiates a transfer on a USB pipe. usbTransferAbort() – Aborts a transfer. usbVendorSpecific() – Allows clients to issue vendor-specific USB requests. usbDataUrbCompleteCallback() – Callback called on URB completion. usbDataVendorSpecificCallback() – Callback called on Vendor Specific Request
DESCRIPTION	Implements the Translation Unit Data Transfer Interfaces.
INCLUDE FILES	usbTransUnit.h , usbHcdInstr.h

usbTransUnitInit

NAME	usbTransUnitInit – Translation Unit Initialization interfaces
ROUTINES	usbInitialize() – Initializes the USB. usbShutdown() – Shuts down the USB. usbClientRegister() – Registers a new client with the USB. usbClientUnregister() – Unregisters a USB client. usbMngmtCallbackSet() – sets a management callback for a client. usbBusStateSet() – Sets bus state, such as SUSPEND or RESUME. usbDynamicAttachRegister() – Registers client for dynamic attach notification. usbDynamicAttachUnRegister() – Unregisters a client for attach notification. usbInitThreadFn() – Translation unit thread routine usbInitClientThreadFn() – Client thread routine usbInitClientIrpCompleteThreadFn() – Client thread routine usbInitDeviceAdd() – Device attach callback usbInitDeviceRemove() – Device detach callback usbInitDeviceSuspend() – Device suspend callback usbInitDeviceResume() – Device resume callback
DESCRIPTION	<p>Implements the translation unit initialization interfaces.</p> <p>In order to use the USB, it is first necessary to invoke usbInitialize(). Multiple calls to usbInitialize() may be nested so long as a corresponding number of calls to usbShutdown() are also made. This allows multiple USB clients to be written</p>

independently and without concern for coordinating the initialization of the independent clients.

Normal USBD clients register with the USBD by calling **usbClientRegister()**. In response to this call, the translation unit allocates per-client data structures and a client callback task. Callbacks for each client are invoked from this client-unique task. This improves the USBD's ability to shield clients from one another and to help ensure a real time response for all clients.

After a client has registered, it will usually register for dynamic attachment notification using **usbDynamicAttachRegister()**. This function allows a special client callback routine to be invoked each time a USB device is attached to or removed from the system. In this way, clients may discover the real time attachment and removal of devices.

INCLUDE FILES **usbTransUnit.h**

usbTransUnitMisc

NAME **usbTransUnitMisc** – translation unit miscellaneous functions

ROUTINES

- usbHcdAttach()** – Attaches an HCD to the USBD.
- usbHcdDetach()** – Detaches an HCD from the USBD.
- usbBusCountGet()** – Gets the number of USBs attached to the host.
- usbRootNodeIdGet()** – Returns the root node for a specific USB.
- usbHubPortCountGet()** – Returns the number of ports connected to a hub.
- usbNodeIdGet()** – Gets the ID of a node connected to a hub port.
- usbAddressGet()** – Gets the USB address for a given device.
- usbAddressSet()** – Sets the USB address for a given device.
- usbVersionGet()** – Returns USBD version information.
- usbStatisticsGet()** – Retrieves USBD operating statistics.
- usbCurrentFrameGet()** – Returns the current frame number for a USB.
- usbNodeInfoGet()** – Returns information about a USB node.

DESCRIPTION This implements translation unit miscellaneous interfaces. These interfaces are used only by **UsbTool** and not by the class drivers. The interfaces are provided to integrate the translation unit with **UsbTool**.

INCLUDE FILES **drv/usb/usbTransUnit.h, usb/pciConstants.h, usb2/usbHubMisc.h, usb2/usbdMisc.h**

usbTransUnitStd

NAME	usbTransUnitStd – translation unit standard requests interfaces
ROUTINES	usbFeatureClear() – Clears a USB feature. usbFeatureSet() – Sets a USB feature. usbConfigurationGet() – Gets the USB configuration for a device. usbConfigurationSet() – Sets the USB configuration for a device. usbDescriptorGet() – Retrieves a USB descriptor. usbDescriptorSet() – Sets a USB descriptor. usbInterfaceGet() – Retrieves the current interface of a device. usbInterfaceSet() – Sets the current interface of a device. usbStatusGet() – Retrieves the USB status from a source such as a device or interface and so on. usbSynchFrameGet() – Returns the isochronous synchronization frame of a device.
DESCRIPTION	Implements the translation unit standard requests interfaces.
INCLUDE FILES	drv/usb/usbTransUnit.h , usb2/usbHcdInstr.h

usbUhcdInitialization

NAME	usbUhcdInitialization – USB UHCI HCD initialization routine
ROUTINES	usbUhcdInstantiate() – instantiate the USB UHCI Host Controller Driver. usbUhcdInit() – initialise the USB UHCI Host Controller Driver. usbUhcdExit() – uninitialize the USB UHCI Host Controller Driver. vxbusUhciRegister() – register the USB UHCI Host Controller Driver with vxBus.
DESCRIPTION	<p>This library defines the entry and exit points for UHCI USB Host Controller Driver. The file initializes the USB host controller Driver. It also exposed routines to initializes the UHCI Controllers.</p> <p>usbHcdUhciDeviceInit() routine implements the legacy support. It handles the hand-off of USB UHCI Controllers from BIOS to system software.</p> <p>usbHcdUhciDeviceConnect() routine initializes the USB UHCI Host Controller and makes it operational to handle USB operations.</p> <p>The implementation of this library follows the UHCI Specification Rev 1.1</p>

INCLUDE FILES `usb2/usbOsal.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhdSupport.h, usb2/usbUhdCommon.h, usb2/usbUhdScheduleQueue.h, usb2/usbUhdScheduleQSupport.h, rebootLib.h`

usbUhdIsr

NAME `usbUhdIsr` – USB UHCI HCD interrupt handler

ROUTINES

DESCRIPTION This file contains the Interrupt Service Routine for the UHCI driver.

INCLUDE FILES `usb2/usbOsal.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhdSupport.h, usb2/usbUhdCommon.h, usb2/usbUhdScheduleQueue.h, usb2/BusAbstractionLayer.h`

usbUhdManagePort

NAME `usbUhdManagePort` – USB UHCI HCD port status handler

ROUTINES

DESCRIPTION This file contains the handlers which regularly scan the UHCI's port for status change

INCLUDE FILES `usb2/usbOsal.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhdCommon.h, usb2/usbUhdScheduleQueue.h, usb2/usbUhdSupport.h, usb2/BusAbstractionLayer.h`

usbUhdRhEmulate

NAME `usbUhdRhEmulate` – USB UHCI HCD Roothub Emulation

ROUTINES

DESCRIPTION This file contains functions which essentially form a wrapper around UHCI's root hub so as to make it appear as an ordinary hub.

INCLUDE FILES `usb2/usbOsai.h, usb2/usbUhcCommon.h, usb2/usbHst.h,`
 `usb2/usbUhcScheduleQueue.h, usb2/usbUhcSupport.h, usb2/BusAbstractionLayer.h,`
 `usb2/usbUhcScheduleQSupport.h, usb2/usbUhci.h`

usbUhcScheduleQSupport

NAME `usbUhcScheduleQSupport` – USB UHCD HCD schedule queue support

ROUTINES

DESCRIPTION This file contains functions which provide support to the Schedule and Queue management module.

INCLUDE FILES `usb2/usbOsai.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhcScheduleQueue.h,`
 `usb2/usbUhcScheduleQSupport.h, usb2/usbUhcCommon.h, usb2/usbUhcSupport.h`

usbUhcScheduleQWaitForSignal

NAME `usbUhcScheduleQWaitForSignal` – USB UHCD HCD ISR support routines

ROUTINES

DESCRIPTION This file contains the handlers that would be invoked by the ISR when relevent interrupts occur.

INCLUDE FILES `usb2/usbOsai.h, usb2/usbHst.h, usb2/usbUhcCommon.h,`
 `usb2/usbUhcScheduleQueue.h, usb2/usbUhcSupport.h,`
 `usb2/usbUhcScheduleQSupport.h, usb2/usbUhci.h`

usbUhcScheduleQueue

NAME `usbUhcScheduleQueue` – USB UHCD HCD schdule queue routines

ROUTINES

DESCRIPTION This file contains functions which are used for transfer scheduling and management.

INCLUDE FILES `usb2/usbOsal.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhcdCommon.h, usb2/usbUhcdScheduleQueue.h, usb2/usbUhcdSupport.h, usb2/usbUhcdScheduleQSupport.h, usb2/usbUhcdRhEmulate.h, usb/usbPciLib.h`

usbUhcdSupport

NAME `usbUhcdSupport` – USB UHCD HCD register access routines

ROUTINES

DESCRIPTION This file contains the functions which would be used to access various register/sub-fields of the UHCD.

INCLUDE FILES `usb/usbOsal.h, usb/usbHst.h, usbUhci.h, usbUhcdScheduleQueue.h, usbUhcdSupport.h, usbUhcdCommon.h, usbUhcdScheduleQueue.h`

usbVxbRegAccess

NAME `usbVxbRegAccess` – library for read/write routines

ROUTINES

`usbRegRead8()` – reads 8-bit USB Register Space
`usbRegRead16()` – reads 16-bit USB Register Space
`usbRegRead32()` – reads 32-bit USB Register Space
`usbRegWrite16()` – writes into 16-bit USB Register Space
`usbRegWrite32()` – writes into 32-bit USB Register Space

DESCRIPTION This file contains of routines which should be used for register reads and writes. The functions uses `vxBus` provided interfaces for register read and write operations. All USB register reads and writes should happen through this library.

INCLUDE FILES `hwif/vxbus/vxBus.h, src/hwif/h/vxbus/vxbAccess.h`

usbd

NAME `usbd` – USB D Routines

usbd

ROUTINES

usbdInit() – initializes USBD2.0
usbdExit() – exits USBD2.0
usbHstDriverRegister() – register class driver
usbHstDriverDeregister() – deregisters USB class driver
usbHstHCDRegister() – register Host Controller Driver with USBD
usbHstHCDDeregister() – deregister a Host Controller Driver
usbHstBusRegister() – registers an USB Bus
usbHstBusDeregister() – deregister a USB Bus
usbVxbRootHubAdd() – configures the root hub
usbVxbRootHubRemove() – removes the root hub

DESCRIPTION

This file initializes the global variables for USB2.0 USBD module and registers itself with the class drivers and host controller driver modules.

Host Controller Driver Registration with USBD - The host controller driver module register themselves with USBD by calling the routine `usbHstHCDRegister()`. The host controller driver is also registered with the `vxBus` as a bus type specifying appropriate `busID`. Subsequent to this routine, the host controller driver calls the routine `usbHstBusRegister()` for every host controller device of the particular HCD. This routine, registers a bus for every host controller device.

Class Driver Registration with USBD - The Class Drivers register with the USBD by calling the routine `usbHstDriverRegister()`. In this routine, the structure **DRIVER_REGISTRATION** is populated and the class driver is in turn registered with `vxBus`.

Device Connection - On a new device notification, the USBD module will call `vxDeviceAnnounce()` to announce the new device. Subsequently `vxBus` will call `usbdDriverFind()` routine to look for a matching driver for the device. If the matching driver is found, its corresponding routine is called to configure the device.

This file includes the **urb.c** and **device.c** source files

INCLUDE FILES

usb2/usbd.h

2

Routines

CmdParserExitFunc() – Terminates parser execution	51
CmdParserHelpFunc() – Displays list of supported commands	51
ExecCmd() – Execute the command line	52
GetHexToken() – Retrieves value of hex token	53
GetNextToken() – Retrieves the next token from an input string	53
KeywordMatch() – Compare keywords	54
PromptAndExecCmd() – Prompt for a command and execute it.	54
SkipSpace() – Skips leading white space in a string	55
TruncSpace() – Truncates string to eliminate trailing whitespace	56
bulkInErpCallbackCSW() – send the CSW on bulk-in pipe	56
bulkInErpCallbackData() – process end of data phase on bulk-in pipe	57
bulkOutErpCallbackCBW() – process the CBW on bulk-out pipe	57
bulkOutErpCallbackData() – process end of data phase on bulk-out pipe	58
ossCalloc() – Allocates memory initialized to zeros.	58
ossFree() – Master USB memory free routine.	59
ossInitialize() – Initializes ossLib .	59
ossMalloc() – Master USB memory allocation routine.	60
ossMemUsedGet() – Retrieves the amount of memory currently in use by USB.	60
ossMutexCreate() – Creates a new mutex.	60
ossMutexDestroy() – Destroys a mutex.	61
ossMutexRelease() – Releases (gives) a mutex.	61
ossMutexTake() – Attempts to take a mutex.	62
ossOldFree() – Frees globally allocated memory.	62
ossOldInstall() – Installs the old method of USB malloc and free.	63
ossOldMalloc() – Global memory allocation	63
ossPartFree() – Frees globally allocated memory.	64
ossPartIdGet() – Retrieves the partition ID of USB memory partition.	64
ossPartMalloc() – USB memory allocation	65
ossPartSizeGet() – Retrieves the size of the USB memory partition.	65
ossPartSizeSet() – Sets the the initial size of the USB memory partition.	66

ossSemCreate() – Creates a new semaphore. 66
ossSemDestroy() – Destroys a semaphore. 67
ossSemGive() – Signals a semaphore. 67
ossSemTake() – Attempts to take a semaphore. 68
ossShutdown() – Shuts down **ossLib**. 68
ossStatus() – Returns **OK** or **ERROR** and sets **errno** based on status. 69
ossThreadCreate() – Spawns a new thread. 69
ossThreadDestroy() – Attempts to destroy a thread. 70
ossThreadSleep() – Voluntarily relinquishes the CPU. 70
ossTime() – Returns the relative system time in msec. 71
pegasusMuxTxRestart() – place muxTxRestart on netJobRing 71
pegasusOutIrpInUse() – determines if any of the output IRP's are in use 72
usbBulkBlkDevCreate() – create a block device 72
usbBulkDevInit() – registers USB Bulk only mass storage class driver 73
usbBulkDevIoctl() – perform a device-specific control 73
usbBulkDevLock() – Marks **USB_BULK_DEV** structure as in use 74
usbBulkDevShow() – shows routine for displaying all LUNs of a device. 74
usbBulkDevShutDown() – shuts down the USB bulk-only class driver 75
usbBulkDevUnlock() – Marks **USB_BULK_DEV** structure as unused. 75
usbBulkDriveEmpty() – routine to check if drive has media inserted. 76
usbBulkDriveShow() – shows routine for displaying one LUN of a device. 76
usbBulkDynamicAttachRegister() – Register SCSI/BULK-ONLY device attach callback. 77
usbBulkDynamicAttachUnregister() – Unregisters SCSI/BULK-ONLY attach callback. 78
usbBulkGetMaxLun() – Return the max LUN number for a device 78
usbBulkShow() – shows routine for displaying all bulk devices. 79
usbCbiUfiBlkDevCreate() – create a block device 79
usbCbiUfiDevInit() – registers USB CBI mass storage class driver for UFI devices 80
usbCbiUfiDevIoctl() – perform a device-specific control. 80
usbCbiUfiDevLock() – Marks **CBI_UFI_DEV** structure as in use 81
usbCbiUfiDevShutDown() – shuts down the USB CBI mass storage class driver 81
usbCbiUfiDevUnlock() – Marks **CBI_UFI_DEV** structure as unused. 82
usbCbiUfiDynamicAttachRegister() – Register UFI device attach callback. 82
usbCbiUfiDynamicAttachUnregister() – Unregisters **CBI_UFI** attach callback. 83
usbConfigCountGet() – Retrieves the number of device configurations. 84
usbConfigDescrGet() – Reads the full configuration descriptor from device. 84
usbDescrCopy() – copies descriptor to a buffer 85
usbDescrCopy32() – copies descriptor to a buffer 86
usbDescrParse() – search a buffer for the a particular USB descriptor 86
usbDescrParseSkip() – search for a descriptor and increment buffer. 87
usbDescrStrCopy() – copies an ASCII string to a string descriptor. 87
usbDescrStrCopy32() – copies an ASCII string to a string descriptor 88
usbEhcdExit() – uninitializes the EHCI Host Controller 88
usbEhcdInit() – initializes the EHCI Host Controller Driver 89
usbEhcdInstantiate() – instantiate the USB EHCI Host Controller Driver. 89
usbEhcdRHCcancelURB() – cancels a request submitted for an endpoint 90

usbEhcdRHDeletePipe() – deletes a pipe specific to an endpoint. 90
usbEhcdRHSubmitURB() – submits a request to an endpoint. 91
usbEhcdRhClearPortFeature() – clears a feature of the port 91
usbEhcdRhCreatePipe() – creates a pipe specific to an endpoint. 92
usbEhcdRhGetHubDescriptor() – get the hub descriptor 93
usbEhcdRhGetPortStatus() – get the status of the port 93
usbEhcdRhProcessClassSpecificRequest() – processes a class specific request 94
usbEhcdRhProcessControlRequest() – processes a control transfer request 94
usbEhcdRhProcessInterruptRequest() – processes a interrupt transfer request 95
usbEhcdRhProcessStandardRequest() – processes a standard transfer request 95
usbEhcdRhSetPortFeature() – set the features of the port 96
usbHalTcdAddressSet() – hal interface to set address. 96
usbHalTcdAttach() – attaches a TCD 97
usbHalTcdCurrentFrameGet() – hal interface to get Current Frame Number. 97
usbHalTcdDetach() – detaches a TCD 98
usbHalTcdDeviceFeatureClear() – hal interface to clear feature on device. 98
usbHalTcdDeviceFeatureSet() – hal interface to set feature on the device. 99
usbHalTcdDisable() – disables the target controller 99
usbHalTcdEnable() – enables the target controller. 100
usbHalTcdEndpointAssign() – configure an endpoint on the target controller 100
usbHalTcdEndpointRelease() – unconfigure endpoint on the target controller 101
usbHalTcdEndpointStateSet() – set the state of an endpoint 101
usbHalTcdEndpointStatusGet() – get the status of an endpoint 102
usbHalTcdErpCancel() – cancel an ERP 102
usbHalTcdErpSubmit() – submit an ERP for an endpoint 103
usbHalTcdSignalResume() – hal interface to initiate resume signal. 103
usbHandleCreate() – Creates a new handle. 104
usbHandleDestroy() – Destroys a handle. 104
usbHandleInitialize() – Initializes the handle utility library. 105
usbHandleShutdown() – Shuts down the handle utility library. 105
usbHandleValidate() – Validates a handle. 106
usbHidIdleSet() – Issues a SET_IDLE request to a USB HID. 106
usbHidProtocolSet() – Issues a SET_PROTOCOL request to a USB HID. 107
usbHidReportSet() – Issues a SET_REPORT request to a USB HID. 107
usbHstBusDeregister() – deregister a USB Bus 108
usbHstBusRegister() – registers an USB Bus 109
usbHstDriverDeregister() – deregisters USB class driver 109
usbHstDriverRegister() – register class driver 110
usbHstHCDDeregister() – deregister a Host Controller Driver 110
usbHstHCDDRegister() – register Host Controller Driver with USBBD 111
usbHubExit() – de-registers and cleans up the USB Hub Class Driver. 111
usbHubInit() – registers USB Hub Class Driver function pointers. 112
usbKeyboardDevInit() – initialize USB keyboard SIO driver 112
usbKeyboardDevShutdown() – shuts down keyboard SIO driver 112
usbKeyboardDynamicAttachRegister() – Register keyboard attach callback 113

usbKeyboardDynamicAttachUnregister()	– Unregisters keyboard attach callback	114
usbKeyboardSioChanLock()	– Marks SIO_CHAN structure as in use	114
usbKeyboardSioChanUnlock()	– Marks SIO_CHAN structure as unused	115
usbListFirst()	– Returns first entry on a linked list.	115
usbListLink()	– Adds an element to a linked list.	116
usbListLinkProt()	– Adds an element to a list guarded by a mutex.	116
usbListNext()	– Retrieves the next pStruct in a linked list.	117
usbListUnlink()	– Removes an entry from a linked list.	117
usbListUnlinkProt()	– Removes an element from a list guarded by a mutex.	118
usbMouseDevInit()	– initialize USB mouse SIO driver	118
usbMouseDevShutdown()	– shuts down mouse SIO driver	119
usbMouseDynamicAttachRegister()	– Register mouse attach callback	119
usbMouseDynamicAttachUnregister()	– Unregisters mouse attach callback	120
usbMouseSioChanLock()	– Marks SIO_CHAN structure as in use	120
usbMouseSioChanUnlock()	– Marks SIO_CHAN structure as unused	121
usbMsBulkInErpInUseFlagGet()	– get the Bulk-in ERP inuse flag	122
usbMsBulkInErpInUseFlagSet()	– set the Bulk-In ERP inuse flag	122
usbMsBulkInErpInit()	– initialize the bulk-in ERP	122
usbMsBulkInStall()	– stall the bulk-in pipe	123
usbMsBulkInUnStall()	– unSTALL the bulk-in pipe	123
usbMsBulkOutErpInUseFlagGet()	– get the Bulk-Out ERP inuse flag	124
usbMsBulkOutErpInUseFlagSet()	– set the Bulk-Out ERP inuse flag	124
usbMsBulkOutErpInit()	– initialize the bulk-Out ERP	125
usbMsBulkOutStall()	– stall the bulk-out pipe	125
usbMsBulkOutUnStall()	– unSTALL the bulk-out pipe	125
usbMsCBWGet()	– get the last mass storage CBW received	126
usbMsCBWInit()	– initialize the mass storage CBW	126
usbMsCSWGet()	– get the current CSW	127
usbMsCSWInit()	– initialize the CSW	127
usbMsIsConfigured()	– test if the device is configured	127
usbMsTestRxCallback()	– invoked after test data is received	128
usbMsTestTxCallback()	– invoked after test data transmitted	128
usbOhcdInit()	– initialize the USB OHCI Host Controller Driver.	129
usbOhciDumpEndpointDescriptor()	– dump endpoint descriptor contents	129
usbOhciDumpGeneralTransferDescriptor()	– dump general transfer descriptor	130
usbOhciDumpMemory()	– dump memory contents	130
usbOhciDumpPendingTransfers()	– dump pending transfers	131
usbOhciDumpPeriodicEndpointList()	– dump periodic endpoint descriptor list	131
usbOhciDumpRegisters()	– dump registers contents.	132
usbOhciExit()	– uninitialize the USB OHCI Host Controller Driver.	132
usbOhciInitializeModuleTestingFunctions()	– obtains entry points	133
usbOhciInstantiate()	– instantiate the USB OHCI Host Controller Driver.	133
usbPegasusDevLock()	– marks USB_PEGASUS_DEV structure as in use	134
usbPegasusDevUnlock()	– marks USB_PEGASUS_DEV structure as unused	134
usbPegasusDynamicAttachRegister()	– register PEGASUS device attach callback	135

usbPegasusDynamicAttachUnregister() – unregisters PEGASUS attach callbackx 136
usbPegasusEndInit() – initializes the pegasus library 136
usbPegasusEndLoad() – initialize the driver and device 137
usbPegasusEndInit() – un-initializes the pegasus class driver 138
usbPegasusReadReg() – read contents of specified and print 138
usbPrinterDevInit() – initialize USB printer SIO driver 139
usbPrinterDevShutdown() – shuts down printer SIO driver 139
usbPrinterDynamicAttachRegister() – Register printer attach callback 140
usbPrinterDynamicAttachUnregister() – Unregisters printer attach callback 140
usbPrinterSioChanLock() – Marks **SIO_CHAN** structure as in use 141
usbPrinterSioChanUnlock() – Marks **SIO_CHAN** structure as unused 141
usbQueueCreate() – Creates an OS-independent queue structure. 142
usbQueueDestroy() – Destroys a queue. 143
usbQueueGet() – Retrieves a message from a queue. 143
usbQueuePut() – Puts a message into a queue. 144
usbRecurringTime() – calculates recurring time for interrupt/isoch transfers. 144
usbRegRead16() – reads 16-bit USB Register Space 145
usbRegRead32() – reads 32-bit USB Register Space 145
usbRegRead8() – reads 8-bit USB Register Space 146
usbRegWrite16() – writes into 16-bit USB Register Space 146
usbRegWrite32() – writes into 32-bit USB Register Space 147
usbSpeakerDevInit() – initialize USB speaker SIO driver 148
usbSpeakerDevShutdown() – shuts down speaker SIO driver 148
usbSpeakerDynamicAttachRegister() – Register speaker attach callback 148
usbSpeakerDynamicAttachUnregister() – Unregisters speaker attach callback 149
usbSpeakerSeqDevLock() – Marks **SEQ_DEV** structure as in use 150
usbSpeakerSeqDevUnlock() – Marks **SEQ_DEV** structure as unused 150
usbTargControlPayloadRcv() – receives data on the default control pipe 151
usbTargControlResponseSend() – sends data to host on the control pipe 152
usbTargControlStatusSend() – sends control transfer status to the host 152
usbTargCurrentFrameGet() – retrieves the current USB frame number 153
usbTargDeviceFeatureClear() – clears a specific feature 153
usbTargDeviceFeatureSet() – sets or enable a specific feature 154
usbTargDisable() – disables a target channel 154
usbTargEnable() – enables target channel onto USB 155
usbTargInitialize() – initializes the USB Target Library 155
usbTargKbdCallbackInfo() – returns **usbTargKbdLib** callback table 156
usbTargKbdInjectReport() – injects a "boot report" 156
usbTargMgmtCallback() – invoked when HAL detects a management event 157
usbTargMsCallbackInfo() – returns **usbTargPrnLib** callback table 157
usbTargPipeCreate() – creates a pipe for communication on an endpoint 158
usbTargPipeDestroy() – destroys an endpoint pipe 159
usbTargPipeStatusGet() – returns the endpoint status 159
usbTargPipeStatusSet() – sets pipe stalled/unstalled status 160
usbTargPrnCallbackInfo() – returns **usbTargPrnLib** callback table 160

usbTargPrnDataInfo()	– returns buffer status/info	161
usbTargPrnDataRestart()	– restarts listening ERP	161
usbTargRbcBlockDevCreate()	– create an RBC BLK_DEV device.	161
usbTargRbcBlockDevGet()	– return opaque pointer to the RBC BLK I/O DEV device	162
usbTargRbcBlockDevSet()	– set the pointer to the RBC BLK I/O DEV device structure.	162
usbTargRbcBufferWrite()	– write micro-code to the RBC device	163
usbTargRbcCacheSync()	– synchronize the cache of the RBC device	163
usbTargRbcCapacityRead()	– read the capacity of the RBC device	164
usbTargRbcFormat()	– format the RBC device	164
usbTargRbcInquiry()	– retrieve inquiry data from the RBC device	165
usbTargRbcModeSelect()	– select the mode parameter page of the RBC device	165
usbTargRbcModeSelect10()	– select the mode parameter page of the RBC device	166
usbTargRbcModeSense()	– retrieve sense data from the RBC device	166
usbTargRbcModeSense10()	– request for mode sense 10 command	167
usbTargRbcPersistentReserveIn()	– send reserve data to the host	167
usbTargRbcPersistentReserveOut()	– reserve resources on the RBC device	168
usbTargRbcPreventAllowRemoval()	– prevent or allow the removal of the RBC device	168
usbTargRbcRead()	– read data from the RBC device	169
usbTargRbcRelease()	– release a resource on the RBC device	169
usbTargRbcRequestSense()	– request sense data from the RBC device	170
usbTargRbcReserve()	– reserve a resource on the RBC device	170
usbTargRbcStartStop()	– start or stop the RBC device	171
usbTargRbcTestUnitReady()	– test if the RBC device is ready	171
usbTargRbcVendorSpecific()	– vendor specific call	172
usbTargRbcVerify()	– verify the last data written to the RBC device	172
usbTargRbcWrite()	– write to the RBC device	173
usbTargSetupErpCallback()	– handles the setup packet	173
usbTargShutdown()	– shutdown the USB target library	174
usbTargSignalResume()	– drives RESUME signalling on USB	174
usbTargTcdAttach()	– to attach the TCD to the target library	175
usbTargTcdDetach()	– detaches a USB target controller driver	175
usbTargTransfer()	– to transfer data through a pipe	176
usbTargTransferAbort()	– cancels a previously submitted USB_ERP	177
usbTcdIsp1582EvalExec()	– single Entry Point for ISP 1582 TCD	177
usbTcdNET2280Exec()	– single Entry Point for NETCHIP 2280 TCD	178
usbTcdPdiusbd12EvalExec()	– single entry point for PDIUSB12 TCD	178
usbTransferTime()	– Calculates the bus time required for a USB transfer.	179
usbUhdExit()	– uninitialize the USB UHCI Host Controller Driver.	179
usbUhdInit()	– initialise the USB UHCI Host Controller Driver.	180
usbUhdInstantiate()	– instantiate the USB UHCI Host Controller Driver.	180
usbVxbRootHubAdd()	– configures the root hub	181
usbVxbRootHubRemove()	– removes the root hub	181
usbdAddressGet()	– Gets the USB address for a given device.	182
usbdAddressSet()	– Sets the USB address for a given device.	182
usbdBusCountGet()	– Gets the number of USBs attached to the host.	183

usbdbusStateSet() – Sets bus state, such as SUSPEND or RESUME. 183
usbdbClientRegister() – Registers a new client with the USBDB. 184
usbdbClientUnregister() – Unregisters a USBDB client. 185
usbdbConfigurationGet() – Gets the USB configuration for a device. 185
usbdbConfigurationSet() – Sets the USB configuration for a device. 186
usbdbCurrentFrameGet() – Returns the current frame number for a USB. 186
usbdbDescriptorGet() – Retrieves a USB descriptor. 187
usbdbDescriptorSet() – Sets a USB descriptor. 188
usbdbDynamicAttachRegister() – Registers client for dynamic attach notification. 189
usbdbDynamicAttachUnregister() – Unregisters a client for attach notification. 191
usbdbExit() – exits USBDB2.0 191
usbdbFeatureClear() – Clears a USB feature. 192
usbdbFeatureSet() – Sets a USB feature. 193
usbdbHcdAttach() – Attaches an HCD to the USBDB. 194
usbdbHcdDetach() – Detaches an HCD from the USBDB. 194
usbdbHubPortCountGet() – Returns the number of ports connected to a hub. 195
usbdbInit() – initializes USBDB2.0 195
usbdbInitialize() – Initializes the USBDB. 196
usbdbInterfaceGet() – Retrieves the current interface of a device. 196
usbdbInterfaceSet() – Sets the current interface of a device. 197
usbdbMngmtCallbackSet() – sets a management callback for a client. 197
usbdbNodeIdGet() – Gets the ID of a node connected to a hub port. 198
usbdbNodeInfoGet() – Returns information about a USB node. 199
usbdbPipeCreate() – Creates a USB pipe for subsequent transfers. 200
usbdbPipeDestroy() – Destroys a USB data transfer pipe. 201
usbdbRootNodeIdGet() – Returns the root node for a specific USB. 202
usbdbShutdown() – Shuts down the USBDB. 202
usbdbStatisticsGet() – Retrieves USBDB operating statistics. 203
usbdbStatusGet() – Retrieves the USB status from a source such as a device or interface and so on. 204
usbdbSynchFrameGet() – Returns the isochronous synchronization frame of a device. 204
usbdbTransfer() – Initiates a transfer on a USB pipe. 205
usbdbTransferAbort() – Aborts a transfer. 207
usbdbVendorSpecific() – Allows clients to issue vendor-specific USB requests. 208
usbdbVersionGet() – Returns USBDB version information. 208
usbtuDataUrbCompleteCallback() – Callback called on URB completion. 209
usbtuDataVendorSpecificCallback() – Callback called on Vendor Specific Request 209
usbtuInitClientIrpCompleteThreadFn() – Client thread routine 210
usbtuInitClientThreadFn() – Client thread routine 210
usbtuInitDeviceAdd() – Device attach callback 211
usbtuInitDeviceRemove() – Device detach callback 211
usbtuInitDeviceResume() – Device resume callback 212
usbtuInitDeviceSuspend() – Device suspend callback 212
usbtuInitThreadFn() – Translation unit thread routine 213
vxbusbEhciRegister() – registers the EHCI Controller with vxBus 213
vxbusbOhciRegister() – registers OHCI driver with vxBus 214

vxUsbUhciRegister() – register the USB UHCI Host Controller Driver with vxBus. **214**

CmdParserExitFunc()

NAME	CmdParserExitFunc() – Terminates parser execution
SYNOPSIS	<pre> UINT16 CmdParserExitFunc (pVOID param, /* Generic parameter passed down */ char **ppCmd, /* Ptr to remainder of cmd line */ FILE *fin, /* stream for input (if any) */ FILE *fout /* stream for output (if any) */) </pre>
DESCRIPTION	Returns RET_OK , causing the parser to return RET_OK to the caller signally normal termination of the parser.
RETURNS	RET_OK
ERRNO	None.
SEE ALSO	cmdParser

CmdParserHelpFunc()

NAME	CmdParserHelpFunc() – Displays list of supported commands
SYNOPSIS	<pre> UINT16 CmdParserHelpFunc (pVOID param, /* Generic parameter passed down */ char **ppCmd, /* Ptr to remainder of cmd line */ FILE *fin, /* stream for input (if any) */ FILE *fout /* stream for output (if any) */) </pre>
DESCRIPTION	Displays the list of commands in the parser command table to <i>fout</i> . When the parser recognizes that this function is about to be executed, it substitutes a pointer to the current CMD_DESCR table in <i>param</i> . If this function is called directly, <i>param</i> should point to a table of CMD_DESCR structures.
RETURNS	RET_CONTINUE
ERRNO	None.

SEE ALSO **cmdParser**

ExecCmd()

NAME **ExecCmd()** – Execute the command line

SYNOPSIS `UINT16 ExecCmd`
 (
 pVOID param, /* Generic parameter for exec funcs */
 char *pCmd, /* Cmd buffer to be parsed/executed */
 FILE *fin, /* Stream for input */
 FILE *fout, /* Stream for output */
 CMD_DESCR *pCmdTable /* CMD_DESCR table */
)

DESCRIPTION Parses and executes the commands in the *pCmd* buffer. I/O - if any - will go to *fin/fout*. The *pCmd* may contain any number of commands separated by **CMD_SEPARATOR**. *pCmdTable* points to an array of **CMD_DESCR** structures defining the command to be recognized by the parser, and *param* is a generic parameter passed down to individual command execution functions.

RETURNS **RET_OK** for normal termination.
 RET_ERROR for program failure.
 RET_CONTINUE if execution should continue.

ERRNO None.

SEE ALSO **cmdParser**

GetHexToken()

NAME **GetHexToken()** – Retrieves value of hex token

SYNOPSIS `char *GetHexToken`
 (
 char *pStr, /* input string */
 long *pToken, /* buffer to receive token value */
 long defVal /* default value */
)

DESCRIPTION	Retrieves the next token from <i>pCmd</i> line, interprets it as a hex value, and stores the result in <i>pToken</i> . If there are no remaining tokens, stores <i>defVal</i> in <i>pToken</i> instead.
RETURNS	Pointer into <i>pStr</i> following end of copied <i>pToken</i>
ERRNO	None.
SEE ALSO	cmdParser

GetNextToken()

NAME	GetNextToken() – Retrieves the next token from an input string
SYNOPSIS	<pre>char *GetNextToken (char *pStr, /* Input string */ char *pToken, /* Bfr to receive token */ UINT16 tokenLen /* Max length of Token bfr */)</pre>
DESCRIPTION	Copies the next token from <i>pStr</i> to <i>pToken</i> . White space before the next token is discarded. Tokens are delimited by white space and by the command separator, CMD_SEPARATOR . No more than <i>tokenLen</i> - 1 characters from <i>pStr</i> will be copied into <i>pToken</i> . <i>tokenLen</i> must be at least one and <i>pToken</i> will be NULL terminated upon return.
RETURNS	Pointer into <i>pStr</i> following end of copied <i>pToken</i> .
ERRNO	None.
SEE ALSO	cmdParser

KeywordMatch()

NAME	KeywordMatch() – Compare keywords
SYNOPSIS	<pre>int KeywordMatch (char *s1, /* string 1 */ </pre>

```
char *s2, /* string 2 */
int len /* max length to compare */
)
```

DESCRIPTION	Compares <i>s1</i> and <i>s2</i> up to <i>len</i> characters, case insensitive. Returns 0 if strings are equal.
NOTE	This function is equivalent to strnicmp() , but that function is not available in all libraries.
RETURNS	0 if <i>s1</i> and <i>s2</i> are the same -n if <i>s1</i> < <i>s2</i> +n if <i>s1</i> > <i>s2</i>
ERRNO	None.
SEE ALSO	cmdParser

PromptAndExecCmd()

NAME	PromptAndExecCmd() – Prompt for a command and execute it.
SYNOPSIS	<pre>UINT16 PromptAndExecCmd (pVOID param, /* Generic parameter for exec funcs */ char *pPrompt, /* Prompt to display */ FILE *fin, /* Input stream */ FILE *fout, /* Output stream */ CMD_DESCR *pCmdTable /* CMD_DESCR table */)</pre>
DESCRIPTION	Displays <i>pPrompt</i> to <i>fout</i> and prompts for input from <i>fin</i> . Then, parses/executes the command. <i>pCmdTable</i> points to an array of CMD_DESCR structures defining the command to be recognized by the parser, and <i>Param</i> is a generic parameter passed down to individual command execution functions.
RETURNS	RET_OK for normal termination RET_ERROR for program failure. RET_CONTINUE if execution should continue.
ERRNO	None.

SEE ALSO **cmdParser**

SkipSpace()

NAME **SkipSpace()** – Skips leading white space in a string

SYNOPSIS

```
char *SkipSpace
(
    char *pStr  /* Input string */
)
```

DESCRIPTION Returns a pointer to the first non-white-space character in *pStr*.

RETURNS Ptr to first non-white-space character in *pStr*

ERRNO None.

SEE ALSO **cmdParser**

TruncSpace()

NAME **TruncSpace()** – Truncates string to eliminate trailing whitespace

SYNOPSIS

```
UINT16 TruncSpace
(
    char *pStr  /* Input string */
)
```

DESCRIPTION Trucates *pStr* to eliminate trailing white space. Returns count of characters left in *pStr* upon return.

RETURNS Number of characters in *pStr* after truncation.

ERRNO None.

SEE ALSO **cmdParser**

bulkInErpCallbackCSW()

NAME	bulkInErpCallbackCSW() – send the CSW on bulk-in pipe
SYNOPSIS	<pre>void bulkInErpCallbackCSW (pVOID erp /* USB_ERP endpoint request packet */)</pre>
DESCRIPTION	This routine sends the CSW (Command Status Wrapper) back to the host following execution of the CBW.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargRbcLib

bulkInErpCallbackData()

NAME	bulkInErpCallbackData() – process end of data phase on bulk-in pipe
SYNOPSIS	<pre>void bulkInErpCallbackData (pVOID erp /* USB_ERP endpoint request packet */)</pre>
DESCRIPTION	This routine is invoked following a data IN phase to the host.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargRbcLib

bulkOutErpCallbackCBW()

NAME	bulkOutErpCallbackCBW() – process the CBW on bulk-out pipe
SYNOPSIS	<pre>void bulkOutErpCallbackCBW (pVOID erp /* USB_ERP endpoint request packet */)</pre>
DESCRIPTION	This routine processes the the CBW (Command Block Wrapper) which is received on the bulk out pipe.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargRbcLib

bulkOutErpCallbackData()

NAME	bulkOutErpCallbackData() – process end of data phase on bulk-out pipe
SYNOPSIS	<pre>void bulkOutErpCallbackData (pVOID erp /* USB_ERP endpoint request packet */)</pre>
DESCRIPTION	This routine is invoked following a data OUT phase from the host.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargRbcLib

ossCalloc()

NAME	ossCalloc() – Allocates memory initialized to zeros.
SYNOPSIS	<pre>pVOID ossCalloc (UINT32 numBytes /* size of buffer to allocate */)</pre>
DESCRIPTION	ossCalloc() uses ossMalloc() to allocate a block of memory and then initializes it to zeros. Memory allocated using this function should be freed using ossFree() .
RETURNS	Pointer to allocated buffer, or NULL
ERRNO	None
SEE ALSO	ossLib

ossFree()

NAME	ossFree() – Master USB memory free routine.
SYNOPSIS	<pre>void ossFree (pVOID bfr)</pre>
DESCRIPTION	ossFree() calls the free routine installed in the global variable <i>ossFreeFuncPtr</i> . This defaults to ossPartFree() , but can be changed by the users to their own defined free routine or to a non-partition method of malloc/free by calling ossOldInstall() .
RETURNS	N/A
ERRNO	None
SEE ALSO	ossLib

ossInitialize()

NAME	ossInitialize() – Initializes ossLib .
SYNOPSIS	<code>STATUS ossInitialize (void)</code>
DESCRIPTION	This routine should be called once at initialization to initialize the ossLib . Calls to this routine may be nested. This permits multiple, independent libraries to use this library without coordinating the use of ossInitialize() and ossShutdown() across the libraries.
RETURNS	OK or ERROR
ERRNO	None
SEE ALSO	ossLib

ossMalloc()

NAME	ossMalloc() – Master USB memory allocation routine.
SYNOPSIS	<pre>void * ossMalloc (UINT32 numBytes)</pre>
DESCRIPTION	ossMalloc() calls the malloc routine installed in the global variable <i>ossMallocFuncPtr</i> . These default to ossPartMalloc() , but can be changed by the users to their own defined malloc routine or to a non-partition method of malloc/free by calling ossOldInstall() .
RETURNS	Pointer to allocated buffer or NULL
ERRNO	None
SEE ALSO	ossLib

ossMemUsedGet()

NAME	ossMemUsedGet() – Retrieves the amount of memory currently in use by USB.
SYNOPSIS	<code>UINT32 ossMemUsedGet (void)</code>
DESCRIPTION	Returns the amount, in bytes, currently being used by USB.
RETURNS	the number of bytes of memory in use.
ERRNO	None
SEE ALSO	ossLib

ossMutexCreate()

NAME	ossMutexCreate() – Creates a new mutex.
SYNOPSIS	<pre>STATUS ossMutexCreate (pMUTEX_HANDLE pMutexHandle /* Handle of newly created mutex */)</pre>
DESCRIPTION	This function creates a new mutex and returns the handle of that mutex in <i>pMutexHandle</i> . The mutex is created in the untaken state.
RETURNS	OK or STATUS
ERRNO	S_ossLib_BAD_PARAMETER S_ossLib_GENERAL_FAULT
SEE ALSO	ossLib

ossMutexDestroy()

NAME	ossMutexDestroy() – Destroys a mutex.
------	----------------------------------------------

SYNOPSIS `STATUS ossMutexDestroy`
 (
 MUTEX_HANDLE mutexHandle /* Handle of mutex to destroy */
)
DESCRIPTION Destroys the mutex *mutexHandle* created by **ossMutexCreate()**.
RETURNS OK or ERROR
ERRNO S_ossLib_GENERAL_FAULT
SEE ALSO ossLib

ossMutexRelease()

NAME **ossMutexRelease()** – Releases (gives) a mutex.
SYNOPSIS `STATUS ossMutexRelease`
 (
 MUTEX_HANDLE mutexHandle /* Mutex to be released */
)
DESCRIPTION Releases the mutex specified by *mutexHandle*. This function will fail if the calling thread is not the owner of the mutex.
RETURNS OK or ERROR
ERRNO S_ossLib_BAD_HANDLE
SEE ALSO ossLib

ossMutexTake()

NAME **ossMutexTake()** – Attempts to take a mutex.
SYNOPSIS `STATUS ossMutexTake`
 (
 MUTEX_HANDLE mutexHandle, /* Mutex to take */
 UINT32 blockFlag /* specifies blocking action */
)

DESCRIPTION	<p>ossMutexTake() attempts to take the specified mutex. The attempt will succeed if the mutex is not owned by any other threads. If a thread attempts to take a mutex which it already owns, the attempt will succeed. <i>blockFlag</i> specifies the blocking behavior. OSS_BLOCK blocks indefinitely waiting for the mutex to be released. OSS_DONT_BLOCK does not block and returns an error if the mutex is not in the released state. Other values of <i>blockFlag</i> are interpreted as a count of milliseconds to wait for the mutex to be released before declaring an error.</p>
RETURNS	OK or ERROR
ERRNO	S_ossLib_BAD_HANDLE S_ossLib_TIMEOUT S_ossLib_GENERAL_FAULT
SEE ALSO	ossLib

ossOldFree()

NAME	ossOldFree() – Frees globally allocated memory.
SYNOPSIS	<pre>void ossOldFree (void * bfr)</pre>
DESCRIPTION	ossOldFree() frees memory allocated by ossMalloc() .
RETURNS	N/A
ERRNO	None
SEE ALSO	ossLib

ossOldInstall()

NAME	ossOldInstall() – Installs the old method of USB malloc and free.
SYNOPSIS	<pre>void ossOldInstall (void)</pre>

DESCRIPTION	Installs the old method of USB malloc and free. This must be called before the call to usbInitialize() .
RETURNS	N/A
ERRNO	None
SEE ALSO	ossLib

ossOldMalloc()

NAME	ossOldMalloc() – Global memory allocation
SYNOPSIS	<pre>void * ossOldMalloc (UINT32 numBytes /* Size of buffer to allocate */)</pre>
DESCRIPTION	ossOldMalloc() allocates a buffer of <i>numBytes</i> in length and returns a pointer to the allocated buffer. The buffer is allocated from a global pool which can be made visible to all processes and drivers in the system. Memory allocated by this function must be freed by calling ossFree() .
RETURNS	Pointer to allocated buffer, or NULL
ERRNO	None
SEE ALSO	ossLib

ossPartFree()

NAME	ossPartFree() – Frees globally allocated memory.
SYNOPSIS	<pre>void ossPartFree (pVOID bfr)</pre>

DESCRIPTION	ossPartFree() frees memory allocated by ossMalloc() .
RETURNS	N/A
ERRNO	None
SEE ALSO	ossLib

ossPartIdGet()

NAME	ossPartIdGet() – Retrieves the partition ID of USB memory partition.
SYNOPSIS	<code>PART_ID ossPartIdGet (void)</code>
DESCRIPTION	Returns the partition ID of the USB memory partition.
RETURNS	The partition ID
ERRNO	None
SEE ALSO	ossLib

ossPartMalloc()

NAME	ossPartMalloc() – USB memory allocation
SYNOPSIS	<pre>void * ossPartMalloc (UINT32 numBytes /* Size of buffer to allocate */)</pre>
DESCRIPTION	ossPartMalloc() allocates a cache-safe buffer of size <i>numBytes</i> out of the USB partition and returns a pointer to this buffer. The buffer is allocated from a local USB partition. The size of this partition defaults to 64k but can be modified to suit the user's needs. This partition will dynamically grow based on additional need. Memory allocated by this function must be freed by calling ossFree() .

RETURNS	Pointer to the allocated buffer, or NULL
ERRNO	None.
SEE ALSO	ossLib

ossPartSizeGet()

NAME	ossPartSizeGet() – Retrieves the size of the USB memory partition.
SYNOPSIS	UINT32 ossPartSizeGet (void)
DESCRIPTION	Returns the size of the USB memory partition.
RETURNS	Size of partition
ERRNO	None
SEE ALSO	ossLib

ossPartSizeSet()

NAME	ossPartSizeSet() – Sets the the initial size of the USB memory partition.
SYNOPSIS	<pre>STATUS ossPartSizeSet (UINT32 numBytes)</pre>
DESCRIPTION	Sets the size of the USB memory partition. This must be called before the first call to ossMalloc. This will set the size that ossMalloc will use for its allocation. Once ossMalloc has been called, the partition size has been already allocated. To add more memory to the USB partition, you must retrieve the USB partition ID and add more memory using the memPartLib routines.
RETURNS	OK or ERROR

ERRNO None

SEE ALSO **ossLib**, **memPartLib**

ossSemCreate()

NAME **ossSemCreate()** – Creates a new semaphore.

SYNOPSIS **STATUS** ossSemCreate

```
(
    UINT32      maxCount,    /* Max count allowed for semaphore */
    UINT32      curCount,    /* initial count for semaphore */
    pSEM_HANDLE pSemHandle   /* newly created semaphore handle */
)
```

DESCRIPTION This function creates a new semaphore and returns the handle of that semaphore in *pSemHandle*. The semaphore's initial count is set to *curCount* and has a maximum count as specified by *maxCount*.

RETURNS OK or ERROR

ERRNO **S_ossLib_BAD_PARAMETER**
 S_ossLib_GENERAL_FAULT

SEE ALSO **ossLib**

ossSemDestroy()

NAME **ossSemDestroy()** – Destroys a semaphore.

SYNOPSIS **STATUS** ossSemDestroy

```
(
    SEM_HANDLE semHandle    /* Handle of semaphore to destroy */
)
```

DESCRIPTION Destroys the semaphore *semHandle* created by **ossSemCreate()**.

RETURNS OK or ERROR

ERRNO S_ossLib_GENERAL_FAULT

SEE ALSO ossLib

ossSemGive()

NAME ossSemGive() – Signals a semaphore.

SYNOPSIS

```
STATUS ossSemGive
(
    SEM_HANDLE semHandle /* semaphore to signal */
)
```

DESCRIPTION This function signals the specified semaphore. A semaphore may have more than one outstanding signal, as specified by the `maxCount` parameter when the semaphore was created by **ossSemCreate()**. While the semaphore is at its maximum count, additional calls to **ossSemSignal** for that semaphore have no effect.

RETURNS OK or ERROR

ERRNO S_ossLib_BAD_HANDLE

SEE ALSO ossLib

ossSemTake()

NAME ossSemTake() – Attempts to take a semaphore.

SYNOPSIS

```
STATUS ossSemTake
(
    SEM_HANDLE semHandle, /* semaphore to take */
    UINT32      blockFlag /* specifies blocking action */
)
```

DESCRIPTION **ossSemTake()** attempts to take the semaphore specified by *semHandle*. *blockFlag* specifies the blocking behavior. **OSS_BLOCK** blocks indefinitely waiting for the semaphore to be signalled. **OSS_DONT_BLOCK** does not block and returns an error if the semaphore is not in the signalled state. Other values of *blockFlag* are interpreted as a count of milliseconds to wait for the semaphore to enter the signalled state before declaring an error.

RETURNS	OK or ERROR
ERRNO	S_ossLib_BAD_HANDLE S_ossLib_TIMEOUT S_ossLib_GENERAL_FAULT
SEE ALSO	ossLib

ossShutdown()

NAME	ossShutdown() – Shuts down ossLib.
SYNOPSIS	STATUS ossShutdown (void)
DESCRIPTION	This routine should be called once at system shutdown if and only if the corresponding call to ossInitialize() was successful.
RETURNS	OK or ERROR
ERRNO	None
SEE ALSO	ossLib

ossStatus()

NAME	ossStatus() – Returns OK or ERROR and sets errno based on status.
SYNOPSIS	STATUS ossStatus (int status)
DESCRIPTION	If <i>status</i> & 0xffff are not equal to zero, this sets errno to the indicated <i>status</i> and returns ERROR. Otherwise, this does not set errno and returns OK.
RETURNS	OK or ERROR

ERRNO Set ERRNO based on status passed in.

SEE ALSO **ossLib**

ossThreadCreate()

NAME **ossThreadCreate()** – Spawns a new thread.

SYNOPSIS

```
STATUS ossThreadCreate
(
    THREAD_PROTOTYPE func,          /* function to spawn as new thread */
    pVOID param,                    /* Parameter to be passed to new thread */
    /*
        UINT16 priority,             /* OSS_PRIORITY_xxxx */
        pCHAR name,                 /* thread name or NULL */
        pTHREAD_HANDLE pThreadHandle /* Handle of newly spawned thread */
    )
```

DESCRIPTION The **ossThreadCreate()** routine creates a new thread which begins execution with the specified *func*. The *param* argument will be passed to *func*. The **ossThreadCreate()** function creates the new thread with a stack of a default size and with no security restrictions--that is, there are no restrictions on the use of the returned *pThreadHandle* by other threads. The newly created thread will execute in the same address space as the calling thread. *priority* specifies the thread's desired priority; in systems which implement thread priorities, as OSS_PRIORITY_xxxx.

RETURNS OK or ERROR

ERRNO S_ossLib_BAD_PARAMETER
 S_ossLib_GENERAL_FAULT

SEE ALSO **ossLib**

ossThreadDestroy()

NAME **ossThreadDestroy()** – Attempts to destroy a thread.

SYNOPSIS STATUS ossThreadDestroy

```
(  
    THREAD_HANDLE threadHandle /* handle of thread to be destroyed */  
)
```

DESCRIPTION	This function attempts to destroy the thread specified by <i>threadHandle</i> .
NOTE	Generally, this function should be called only after the given thread has terminated normally. Destroying a running thread may result in a failure to release resources allocated by the thread.
RETURNS	OK or ERROR
ERRNO	S_ossLib_GENERAL_FAULT
SEE ALSO	ossLib

ossThreadSleep()

NAME	ossThreadSleep() – Voluntarily relinquishes the CPU.
SYNOPSIS	<pre>STATUS ossThreadSleep (UINT32 msec /* Number of msec to sleep */)</pre>
DESCRIPTION	Threads may call ossThreadSleep() to voluntarily release the CPU to another thread or process. If the <i>msec</i> argument is 0, then the thread will be rescheduled for execution as soon as possible. If the <i>msec</i> argument is greater than 0, then the current thread will sleep for at least the number of milliseconds specified.
RETURNS	OK or ERROR
ERRNO	None
SEE ALSO	ossLib

ossTime()

NAME	ossTime() – Returns the relative system time in msec.
------	--------------------------------------------------------------

SYNOPSIS	UINT32 ossTime (void)
DESCRIPTION	Returns a count of milliseconds relative to the time the system was started.
NOTE	The time will wrap about every 49 days, so time calucations should always be based on the difference between two time values.
RETURNS	relative system time in msec
ERRNO	None
SEE ALSO	ossLib

pegasusMuxTxRestart()

NAME	pegasusMuxTxRestart() – place muxTxRestart on netJobRing
SYNOPSIS	<pre>void pegasusMuxTxRestart (END_OBJ * pEndObj /* pointer to DRV_CTRL structure */)</pre>
DESCRIPTION	This function places the muxTxRestart on netJobRing
RETURNS	N/A
ERRNO	none
SEE ALSO	usbPegasusEnd

pegasusOutIrpInUse()

NAME	pegasusOutIrpInUse() – determines if any of the output IRP's are in use
SYNOPSIS	<pre>BOOL pegasusOutIrpInUse (PEGASUS_DEVICE * pDevCtrl)</pre>

DESCRIPTION	This function determines if any of the output IRP's are in use and returns the status information
RETURNS	TRUE if any of the IRP's are in use, FALSE otherwise.
ERRNO	none
SEE ALSO	usbPegasusEnd

usbBulkBlkDevCreate()

NAME **usbBulkBlkDevCreate()** – create a block device

SYNOPSIS XBD * usbBulkBlkDevCreate

```
(
    USBD_NODE_ID nodeId,      /* nodeId of the bulk-only device */
    UINT8         lun,        /* Logical Unit Number */
    UINT32        numBlks,    /* number of logical blocks on device */
    UINT32        blkOffset,  /* offset of the starting block */
    UINT32        flags       /* optional flags */
)
```

DESCRIPTION This routine initializes a XBD structure, which describes a logical partition on a **USB_BULK_DEV** device. A logical partition is an array of contiguously addressed blocks; it can be completely described by the number of blocks and the address of the first block in the partition.

NOTE If **numBlocks** is 0, the rest of device is used.

This routine supplies an additional parameter called *flags*. This bitfield currently only uses bit 1. This bit determines whether the driver will use a SCSI READ6 or SCSI READ10 for read access.

RETURNS A pointer to the XBD, or NULL if parameters exceed physical device boundaries, or if no bulk device exists.

ERRNO none

SEE ALSO **usbBulkDevLib**

usbBulkDevInit()

NAME	usbBulkDevInit() – registers USB Bulk only mass storage class driver
SYNOPSIS	STATUS usbBulkDevInit (void)
DESCRIPTION	This routine registers the mass storage class driver with USB driver. It also registers attach callback routine to get notified of the USB/MSC/BULK ONLY devices.
RETURNS	OK, or ERROR if unable to register with USBD.
ERRNO	S_usbbulkDevLib_OUT_OF_RESOURCES Resources not available S_usbbulkDevLib_USBD_FAULT Error in USBD layer
SEE ALSO	usbBulkDevLib

usbBulkDevIoctl()

NAME	usbBulkDevIoctl() – perform a device-specific control
SYNOPSIS	<pre>int usbBulkDevIoctl (XBD * pUsbBulkXbdDev, /* pointer to bulk device */ int request, /* request type */ void * someArg /* arguments related to request */)</pre>
DESCRIPTION	Typically called to invoke device-specific functions which are not needed by a file system.
RETURNS	The status of the request, or ERROR if the request is unsupported.
ERRNO	none
SEE ALSO	usbBulkDevLib

usbBulkDevLock()

NAME	usbBulkDevLock() – Marks USB_BULK_DEV structure as in use
SYNOPSIS	<pre>STATUS usbBulkDevLock (USBD_NODE_ID nodeId /* NodeId of the XBD to be marked as in use */)</pre>
DESCRIPTION	A caller uses usbBulkDevLock() to notify usbBulkDevLib that it is using the indicated USB_BULK_DEV structure. usbBulkDevLib maintains a count of callers using a particular USB_BULK_DEV structure so that it knows when it is safe to dispose of a structure when the underlying USB_BULK_DEV is removed from the system. So long as the "lock count" is greater than zero, usbBulkDevLib will not dispose of an USB_BULK_DEV structure.
RETURNS	OK, or ERROR if unable to mark USB_BULK_DEV structure in use
ERRNO	none
SEE ALSO	usbBulkDevLib

usbBulkDevShow()

NAME	usbBulkDevShow() – shows routine for displaying all LUNs of a device.
SYNOPSIS	<pre>void usbBulkDevShow (USBD_NODE_ID nodeId /* nodeId of the bulk-only device */)</pre>
DESCRIPTION	This function displays all the logical unit number of the device specified by <i>nodeId</i>
RETURNS	N/A
ERRNO	none
SEE ALSO	usbBulkDevLib

usbBulkDevShutDown()

NAME	usbBulkDevShutDown() – shuts down the USB bulk-only class driver
SYNOPSIS	<pre>STATUS usbBulkDevShutDown (int errCode /* Error code - reason for shutdown */)</pre>
DESCRIPTION	This routine unregisters the driver from USBSD and releases any resources allocated for the devices.
RETURNS	OK or ERROR depending on errCode
ERRNO	S_usbBulkDevLib_NOT_INITIALIZED Not initialized
SEE ALSO	usbBulkDevLib

usbBulkDevUnlock()

NAME	usbBulkDevUnlock() – Marks USB_BULK_DEV structure as unused.
SYNOPSIS	<pre>STATUS usbBulkDevUnlock (USBSD_NODE_ID nodeId /* NodeId of the XBD to be marked as unused */)</pre>
DESCRIPTION	This function releases a lock placed on an USB_BULK_DEV structure. When a caller no longer needs an USB_BULK_DEV structure for which it has previously called usbBulkDevLock() , then it should call this function to release the lock.
NOTE	If the underlying SCSI/BULK-ONLY device has already been removed from the system, then this function will automatically dispose of the USB_BULK_DEV structure if this call removes the last lock on the structure. Therefore, a caller must not reference the USB_BULK_DEV structure after making this call.
RETURNS	OK, or ERROR if unable to mark USB_BULK_DEV structure unused
ERRNO	S_usbBulkDevLib_NOT_LOCKED No Lock to Unlock

SEE ALSO **usbBulkDevLib**

usbBulkDriveEmpty()

NAME **usbBulkDriveEmpty()** – routine to check if drive has media inserted.

SYNOPSIS

```
BOOL usbBulkDriveEmpty
(
    USBD_NODE_ID nodeId, /* nodeId of the bulk-only device */
    UINT8        lun
)
```

DESCRIPTION This routine simply returns the Empty flag for the drive from the usbBulk structure.

RETURNS TRUE if drive is Empty, FALSE if there is media in the drive

ERRNO none

SEE ALSO **usbBulkDevLib**

usbBulkDriveShow()

NAME **usbBulkDriveShow()** – shows routine for displaying one LUN of a device.

SYNOPSIS

```
void usbBulkDriveShow
(
    USBD_NODE_ID nodeId, /* nodeId of the bulk-only device */
    UINT8        lun
)
```

DESCRIPTION This function displays the device with logical unit number specified as *lun*

RETURNS N/A

ERRNO none

SEE ALSO **usbBulkDevLib**

usbBulkDynamicAttachRegister()

NAME **usbBulkDynamicAttachRegister()** – Register SCSI/BULK-ONLY device attach callback.

SYNOPSIS

```
STATUS usbBulkDynamicAttachRegister
(
    USB_BULK_ATTACH_CALLBACK callback, /* new callback to be registered */
    pVOID arg /* user-defined arg to callback */
)
```

DESCRIPTION *callback* is a caller-supplied function of the form:

```
typedef (*USB_BULK_ATTACH_CALLBACK)
(
    pVOID arg,
    USBD_NODE_ID bulkDevId,
    UINT16 attachCode
);
```

usbBulkDevLib will invoke *callback* each time a MSC/SCSI/BULK-ONLY device is attached to or removed from the system. *arg* is a caller-defined parameter which will be passed to the *callback* each time it is invoked. The *callback* will also be passed the *nodeID* of the device being created/destroyed and an attach code of **USB_BULK_ATTACH** or **USB_BULK_REMOVE**.

NOTE The user callback routine should not invoke any driver function that submits IRPs. Further processing must be done from a different task context. As the driver routines wait for IRP completion, they cannot be invoked from USB client task's context created for this driver.

RETURNS OK, or **ERROR** if unable to register callback

ERRNO

S_usbBulkDevLib_BAD_PARAM
 Bad Parameters passed

S_usbBulkDevLib_OUT_OF_MEMORY
 System Out of Memory

SEE ALSO **usbBulkDevLib**

usbBulkDynamicAttachUnregister()

NAME **usbBulkDynamicAttachUnregister()** – Unregisters SCSI/BULK-ONLY attach callback.

SYNOPSIS

```
STATUS usbBulkDynamicAttachUnregister
```

```
(
    USB_BULK_ATTACH_CALLBACK callback, /* callback to be unregistered */
    pVOID arg /* user-defined arg to callback */
)
```

DESCRIPTION	This function cancels a previous request to be dynamically notified for SCSI/BULK-ONLY device attachment and removal. The <i>callback</i> and <i>arg</i> parameters must exactly match those passed in a previous call to usbBulkDynamicAttachRegister() .
RETURNS	OK, or ERROR if unable to unregister callback
ERRNO	S_usbBulkDevLib_NOT_REGISTERED Could not register the callback
SEE ALSO	usbBulkDevLib

usbBulkGetMaxLun()

NAME	usbBulkGetMaxLun() – Return the max LUN number for a device
SYNOPSIS	<pre>UINT8 usbBulkGetMaxLun (USBD_NODE_ID nodeId /* nodeId of the bulk-only device */)</pre>
DESCRIPTION	This function returns the maximum LUN number of the device
RETURNS	UINT8 value specifying the maximum LUN or 0, if <i>nodeId</i> not found
ERRNO	none
SEE ALSO	usbBulkDevLib

usbBulkShow()

NAME	usbBulkShow() – shows routine for displaying all bulk devices.
SYNOPSIS	<pre>void usbBulkShow ()</pre>

DESCRIPTION	This routine displays all the bulk devices connected
RETURNS	N/A
ERRNO	none
SEE ALSO	usbBulkDevLib

usbCbiUfiBlkDevCreate()

NAME	usbCbiUfiBlkDevCreate() – create a block device
SYNOPSIS	<pre>XBD * usbCbiUfiBlkDevCreate (USBD_NODE_ID nodeId /* Node Id of the CBI_UFI device */)</pre>
DESCRIPTION	This routine initializes a XBD structure, which describes a logical partition on a USB_CBI_UFI_DEV device. A logical partition is an array of contiguously addressed blocks; it can be completely described by the number of blocks and the address of the first block in the partition.
RETURNS	A pointer to the XBD, or NULL if no CBI/UFI device exists.
ERRNO	none
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDevInit()

NAME	usbCbiUfiDevInit() – registers USB CBI mass storage class driver for UFI devices
SYNOPSIS	<pre>STATUS usbCbiUfiDevInit (void)</pre>
DESCRIPTION	This routine registers the CBI mass storage class driver for UFI devices. It also registers a callback routine to request notification whenever USB/MSC/CBI/UFI devices are attached or removed.
RETURNS	OK, or ERROR if unable to register with USBSD.

ERRNO	S_usbCbiUfiDevLib_OUT_OF_RESOURCES Resources are not available
	S_usbCbiUfiDevLib_USBD_FAULT USB Fault has occurred
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDevIoctl()

NAME	usbCbiUfiDevIoctl() – perform a device-specific control.
-------------	-----------------------------------------------------------------

SYNOPSIS	<pre>int usbCbiUfiDevIoctl (XBD * pCbiUfiXbdDev, /* pointer to MSC/CBI/UFI device */ int request, /* request type */ void * someArg /* arguments related to request */)</pre>
-----------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

DESCRIPTION	Typically called by file system to invoke device-specific functions beyond file handling. The following control requests are supported
	FIODISKFORMAT (0x05) Formats the entire disk with appropriate hardware track and sector marks. No file system is initialized on the disk by this request. This control function is defined by the file system, but provided by the driver.
	USB UFI ALL DESCRIPTOR GET (0xF0) Invokes show routine for displaying configuration, device and interface descriptors.
	USB UFI DEV RESET (0xF1) Issues a command block reset and clears stall condition on bulk-in and bulk-out endpoints.
RETURNS	The status of the request, or ERROR if the request is unsupported.
ERRNO	none
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDevLock()

NAME	usbCbiUfiDevLock() – Marks CBI_UFI_DEV structure as in use
SYNOPSIS	<pre>STATUS usbCbiUfiDevLock (USBD_NODE_ID nodeId /* NodeId of the XBD to be marked as in use */)</pre>
DESCRIPTION	A caller uses usbCbiUfiDevLock() to notify usbCbiUfiDevLib that it is using the indicated CBI_UFI_DEV structure. usbCbiUfiDevLib maintains a count of callers using a particular CBI_UFI_DEV structure so that it knows when it is safe to dispose of a structure when the underlying CBI_UFI_DEV is removed from the system. So long as the "lock count" is greater than zero, usbCbiUfiDevLib will not dispose of an CBI_UFI_DEV structure.
RETURNS	OK, or ERROR if unable to mark CBI_UFI_DEV structure in use
ERRNO	none
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDevShutDown()

NAME	usbCbiUfiDevShutDown() – shuts down the USB CBI mass storage class driver
SYNOPSIS	<pre>STATUS usbCbiUfiDevShutDown (int errCode /* Error code - reason for shutdown */)</pre>
DESCRIPTION	This routine unregisters UFI driver from USBD and releases any resources allocated for the devices.
RETURNS	OK or ERROR.
ERRNO	S_usbCbiUfiDevLib_NOT_INITIALIZED CBI Device is not initialized
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDevUnlock()

NAME	usbCbiUfiDevUnlock() – Marks CBI_UFI_DEV structure as unused.
SYNOPSIS	<pre>STATUS usbCbiUfiDevUnlock (USBD_NODE_ID nodeId /* NodeId of the XBD to be marked as unused */)</pre>
DESCRIPTION	This function releases a lock placed on an CBI_UFI_DEV structure. When a caller no longer needs an CBI_UFI_DEV structure for which it has previously called usbCbiUfiDevLock() , then it should call this function to release the lock.
NOTE	If the underlying CBI_UFI device has already been removed from the system, then this function will automatically dispose of the CBI_UFI_DEV structure if this call removes the last lock on the structure. Therefore, a caller must not reference the CBI_UFI_DEV structure after making this call.
RETURNS	OK, or ERROR if unable to mark CBI_UFI_DEV structure unused
ERRNO	S_usbCbiUfiDevLib_NOT_LOCKED No lock to Unlock
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDynamicAttachRegister()

NAME	usbCbiUfiDynamicAttachRegister() – Register UFI device attach callback.
SYNOPSIS	<pre>STATUS usbCbiUfiDynamicAttachRegister (USB_UFI_ATTACH_CALLBACK callback, /* new callback to be registered */ PVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre>typedef (*USB_UFI_ATTACH_CALLBACK) (PVOID arg, USBD_NODE_ID cbiUfiDevId, UINT16 attachCode);</pre>

usbCbiUfiDevLib will invoke *callback* each time a **CBI_UFI** device is attached to or removed from the system. *arg* is a caller-defined parameter which will be passed to the *callback* each time it is invoked. The *callback* will also be passed the *nodeID* of the device being created/destroyed and an attach code of **USB_UFI_ATTACH** or **USB_UFI_REMOVE**.

NOTE	The user callback routine should not invoke any driver function that submits IRPs. Further processing must be done from a different task context. As the driver routines wait for IRP completion, they cannot be invoked from USBBD client task's context created for this driver.
RETURNS	OK, or ERROR if unable to register callback
ERRNO	S_usbCbiUfiDevLib_BAD_PARAM Bad Paramter passed S_usbCbiUfiDevLib_OUT_OF_MEMORY Sufficient memory not available
SEE ALSO	usbCbiUfiDevLib

usbCbiUfiDynamicAttachUnregister()

NAME	usbCbiUfiDynamicAttachUnregister() – Unregisters CBI_UFI attach callback.
SYNOPSIS	<pre>STATUS usbCbiUfiDynamicAttachUnregister (USB_UFI_ATTACH_CALLBACK callback, /* callback to be unregistered */ pVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	This function cancels a previous request to be dynamically notified for CBI_UFI device attachment and removal. The <i>callback</i> and <i>arg</i> paramters must exactly match those passed in a previous call to usbCbiUfiDynamicAttachRegister() .
RETURNS	OK, or ERROR if unable to unregister callback
ERRNO	S_usbCbiUfiDevLib_NOT_REGISTERED Could not register the callback
SEE ALSO	usbCbiUfiDevLib

usbConfigCountGet()

NAME	usbConfigCountGet() – Retrieves the number of device configurations.
SYNOPSIS	<pre>STATUS usbConfigCountGet (USBD_CLIENT_HANDLE usbdcClientHandle, /* caller's USBD client handle */ USBD_NODE_ID nodeId, /* device node ID */ pUINT16 pNumConfig /* bfr to receive nbr of config */)</pre>
DESCRIPTION	Using the <i>usbdcClientHandle</i> provided by the caller, this function reads the <i>nodeId</i> 's device descriptor and returns the number of configurations supported by the device in <i>pNumConfig</i> .
RETURNS	OK, or ERROR if unable to read device descriptor
ERRNO	None
SEE ALSO	usbLib

usbConfigDescrGet()

NAME	usbConfigDescrGet() – Reads the full configuration descriptor from device.
SYNOPSIS	<pre>STATUS usbConfigDescrGet (USBD_CLIENT_HANDLE usbdcClientHandle, /* caller's USBD client handle */ USBD_NODE_ID nodeId, /* device node ID */ UINT16 cfgNo, /* specifies configuration nbr */ pUINT16 pBfrLen, /* receives length of buffer */ pUINT8 *ppBfr /* receives pointer to buffer */)</pre>
DESCRIPTION	<p>This function reads the configuration descriptor <i>cfgNo</i> and all associated descriptors (interface, endpoint, and so on) for the device specified by <i>nodeId</i>. The total amount of data returned by a device is variable, so this function pre-reads just the configuration descriptor and uses the "totalLength" field from that descriptor to determine the total length of the configuration descriptor and its associated descriptors.</p> <p>This function uses the macro OSS_MALLOC() to allocate a buffer for the complete descriptor. The size and location of the buffer are returned in <i>ppBfr</i> and <i>pBfrLen</i>. It is the caller's responsibility to free the buffer using the OSS_FREE() macro.</p>

RETURNS OK, or **ERROR** if unable to read descriptor

ERRNO None

SEE ALSO **usbLib**

usbDescrCopy()

NAME **usbDescrCopy()** – copies descriptor to a buffer

SYNOPSIS

```
VOID usbDescrCopy
(
    pUINT8  pBfr,      /* destination buffer */
    pVOID   pDescr,    /* source buffer */
    UINT16  bfrLen,    /* dest len */
    pUINT16 pActLen    /* actual length copied */
)
```

DESCRIPTION Copies the USB descriptor at *pDescr* to the *pBfr* of length *bfrLen*. Returns the actual number of bytes copied - which is the shorter of the *pDescr* or *bfrLen* - in *pActLen* if *pActLen* is non-NULL.

RETURNS N/A

ERRNO None

SEE ALSO **usbDescrCopyLib**

usbDescrCopy32()

NAME **usbDescrCopy32()** – copies descriptor to a buffer

SYNOPSIS

```
VOID usbDescrCopy32
(
    pUINT8  pBfr,      /* destination buffer */
    pVOID   pDescr,    /* source buffer */
    UINT32  bfrLen,    /* dest len */
    pUINT32 pActLen    /* actual length copied */
)
```

DESCRIPTION	This function is the same as usbDescrCopy() except that <i>bfrLen</i> and <i>pActLen</i> refer to UINT32 quantities.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbDescrCopyLib

usbDescrParse()

NAME	usbDescrParse() – search a buffer for the a particular USB descriptor
SYNOPSIS	<pre>pVOID usbDescrParse (pUINT8 pBfr, /* buffer to parse */ UINT16 bfrLen, /* length of buffer to parse */ UINT8 descriptorType /* type of descriptor being sought */)</pre>
DESCRIPTION	Searches <i>pBfr</i> up to <i>bfrLen</i> bytes for a descriptor of a type matching <i>descriptorType</i> and returns a pointer to the descriptor if found.
RETURNS	pointer to indicated descriptor, or NULL if descr not found
ERRNO	None
SEE ALSO	usbLib

usbDescrParseSkip()

NAME	usbDescrParseSkip() – search for a descriptor and increment buffer.
SYNOPSIS	<pre>pVOID usbDescrParseSkip (pUINT8 *ppBfr, /* buffer to parse */ pUINT16 pBfrLen, /* length of buffer to parse */ UINT8 descriptorType /* type of descriptor being sought */)</pre>

DESCRIPTION	This searches <i>ppBfr</i> up to <i>pBfrLen</i> bytes for a descriptor of a type matching <i>descriptorType</i> and returns a pointer to the descriptor if found. <i>ppBfr</i> and <i>pBfrLen</i> are updated to reflect the next location in the buffer and the remaining size of the buffer, respectively.
RETURNS	pointer to indicated descriptor, or NULL if descr not found.
ERRNO	None
SEE ALSO	usbLib

usbDescrStrCopy()

NAME	usbDescrStrCopy() – copies an ASCII string to a string descriptor.
SYNOPSIS	<pre> VOID usbDescrStrCopy (pUINT8 pBfr, /* destination buffer */ char *pStr, /* source buffer */ UINT16 bfrLen, /* dest len */ pUINT16 pActLen /* actual length copied */) </pre>
DESCRIPTION	This routine constructs a properly formatted USB string descriptor in <i>pBfr</i> . The ASCII string <i>pStr</i> is copied to <i>pBfr</i> as a Unicode string as required by the USB spec. The actual length of the resulting descriptor is returned in <i>pActLen</i> if <i>pActLen</i> is non-NULL.
NOTE	The complete length of the string descriptor can be calculated as $2 * \text{strlen}(pStr) + 2$. The <i>pActLen</i> will be the shorter of <i>bfrLen</i> or this value.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbDescrCopyLib

usbDescrStrCopy32()

NAME	usbDescrStrCopy32() – copies an ASCII string to a string descriptor
-------------	----------------------------------------------------------------------------

SYNOPSIS	<pre>VOID usbDescrStrCopy32 (PUINT8 pBfr, /* destination buffer */ char *pStr, /* source buffer */ UINT32 bfrLen, /* dest len */ PUINT32 pActLen /* actual length copied */)</pre>
DESCRIPTION	This function is the same as usbDescrStrCopy() except that <i>bfrLen</i> and <i>pActLen</i> refer to UINT32 quantities.
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbDescrCopyLib

usbEhcdExit()

NAME	usbEhcdExit() – uninitializes the EHCI Host Controller
SYNOPSIS	<pre>BOOLEAN usbEhcdExit(void)</pre>
DESCRIPTION	This routine uninitializes the EHCI Host Controller Driver and detaches it from the usbd interface layer.
RETURNS	TRUE, or FALSE if there is an error during HCD uninitialization.
ERRNO	None.
SEE ALSO	usbEhcdInitExit

usbEhcdInit()

NAME	usbEhcdInit() – initializes the EHCI Host Controller Driver
SYNOPSIS	<pre>STATUS usbEhcdInit (void)</pre>

DESCRIPTION	<p>This routine initializes the EHCI Host Controller Driver data structures. This routine is executed prior to vxBus device connect to initialize data structures expected by the device initialization.</p> <p>The USB D must be initialized prior to calling this routine. In this routine the book-keeping variables for the EHCI Driver are initialized.</p> <p>The function also registers the EHCI Host controller Drive with USB D</p>
RETURNS	OK or ERROR, if the initialization fails
ERRNO	None.
SEE ALSO	usbEhcdInitExit

usbEhcdInstantiate()

NAME	usbEhcdInstantiate() – instantiate the USB EHCI Host Controller Driver.
SYNOPSIS	VOID usbEhcdInstantiate (void)
DESCRIPTION	<p>This routine instantiates the EHCI Host Controller Driver and allows the EHCI Controller driver to be included with the vxWorks image and not be registered with vxBus. EHCI devices will remain orphan devices until the usbEhciInit() routine is called. This supports the INCLUDE_EHCI behaviour of previous vxWorks releases.</p> <p>The routine itself does nothing.</p>
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbEhcdInitExit

usbEhcdRHCancelURB()

NAME	usbEhcdRHCancelURB() – cancels a request submitted for an endpoint
-------------	---------------------------------------------------------------------

SYNOPSIS	<pre>USBHST_STATUS usbEhcdRHCcancelURB (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ UINT32 uPipeHandle, /* Pipe Handle Identifier */ pUSBHST_URB pURB /* Ptr to User Request Block */)</pre>
DESCRIPTION	This routine cancels a request submitted for an endpoint.
RETURNS	USBHST_SUCCESS if the URB is submitted successfully. USBHST_INVALID_PARAMETER if the parameters are not valid. USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRHDeletePipe()

NAME	usbEhcdRHDeletePipe() – deletes a pipe specific to an endpoint.
SYNOPSIS	<pre>USBHST_STATUS usbEhcdRHDeletePipe (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ UINT32 uPipeHandle /* Pipe Handle Identifier */)</pre>
DESCRIPTION	This routine deletes a pipe specific to an endpoint.
RETURNS	USBHST_SUCCESS if the pipe was deleted successfully. USBHST_INVALID_PARAMETER if the parameters are not valid.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRHSubmitURB()

NAME	usbEhcdRHSubmitURB() – submits a request to an endpoint.
SYNOPSIS	<pre>USBHST_STATUS usbEhcdRHSubmitURB (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ UINT32 uPipeHandle, /* Pipe Handle Identifier */ pUSBHST_URB pURB /* Ptr to User Request Block */)</pre>
DESCRIPTION	This routine submits a request to an endpoint.
RETURNS	USBHST_SUCCESS if the URB is submitted successfully. USBHST_INVALID_PARAMETER if the parameters are not valid. USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRhClearPortFeature()

NAME	usbEhcdRhClearPortFeature() – clears a feature of the port
SYNOPSIS	<pre>USBHST_STATUS usbEhcdRhClearPortFeature (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ pUSBHST_URB pURB /* Ptr to User Request Block */)</pre>
DESCRIPTION	This routine clears a feature of the port.
RETURNS	USBHST_SUCCESS - if the URB is submitted successfully. USBHST_INVALID_PARAMETER - if the parameters are not valid.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRhCreatePipe()

NAME	usbEhcdRhCreatePipe() – creates a pipe specific to an endpoint.
SYNOPSIS	<pre>USBHST_STATUS usbEhcdRhCreatePipe (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ UINT8 uDeviceAddress, /* Device Address */ UINT8 uDeviceSpeed, /* Device Speed */ UCHAR *pEndpointDescriptor, /* Ptr to EndPoint Descriptor */ UINT32 *puPipeHandle /* Ptr to pipe handle */)</pre>
DESCRIPTION	This routine creates a pipe specific to an endpoint.
RETURNS	USBHST_SUCCESS - if the pipe was created successfully. USBHST_INVALID_PARAMETER if the parameters are not valid. USBHST_INSUFFICIENT_MEMORY if the memory allocation for the pipe failed.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRhGetHubDescriptor()

NAME	usbEhcdRhGetHubDescriptor() – get the hub descriptor
SYNOPSIS	<pre>USBHST_STATUS usbEhcdRhGetHubDescriptor (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ pUSBHST_URB pURB /* Ptr to User Request Block */)</pre>
DESCRIPTION	This routine gets the hub descriptor.
RETURNS	USBHST_SUCCESS - if the URB is submitted successfully. USBHST_INVALID_PARAMETER - if the parameters are not valid.
ERRNO	None.

SEE ALSO **usbEhcdRhEmulation**

usbEhcdRhGetPortStatus()

NAME **usbEhcdRhGetPortStatus()** – get the status of the port

SYNOPSIS `USBHST_STATUS usbEhcdRhGetPortStatus
 (
 pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */
 pUSBHST_URB pURB /* Ptr to User Request Block */
)`

DESCRIPTION This routine gets the status of the port.

RETURNS **USBHST_SUCCESS** - if the URB is submitted successfully.
 USBHST_INVALID_PARAMETER - if the parameters are not valid.

ERRNO None.

SEE ALSO **usbEhcdRhEmulation**

usbEhcdRhProcessClassSpecificRequest()

NAME **usbEhcdRhProcessClassSpecificRequest()** – processes a class specific request

SYNOPSIS `USBHST_STATUS usbEhcdRhProcessClassSpecificRequest
 (
 pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */
 pUSBHST_URB pURB /* Ptr to User Request Block */
)`

DESCRIPTION This routine processes a class specific request.

RETURNS **USBHST_SUCCESS** if the URB is submitted successfully.
 USBHST_INVALID_PARAMETER if the parameters are not valid.
 USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.

ERRNO None.

SEE ALSO **usbEhcdRhEmulation**

usbEhcdRhProcessControlRequest()

NAME **usbEhcdRhProcessControlRequest()** – processes a control transfer request

SYNOPSIS USBHST_STATUS usbEhcdRhProcessControlRequest
 (
 pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */
 pUSBHST_URB pURB /* Ptr to User Request Block */
)

DESCRIPTION This routine processes a control transfer request.

RETURNS **USBHST_SUCCESS** if the URB is submitted successfully.
 USBHST_INVALID_PARAMETER if the parameters are not valid.
 USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.

ERRNO None.

SEE ALSO **usbEhcdRhEmulation**

usbEhcdRhProcessInterruptRequest()

NAME **usbEhcdRhProcessInterruptRequest()** – processes a interrupt transfer request

SYNOPSIS USBHST_STATUS usbEhcdRhProcessInterruptRequest
 (
 pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */
 pUSBHST_URB pURB /* Ptr to User Request Block */
)

DESCRIPTION This routine processes a interrupt transfer request.

RETURNS	USBHST_SUCCESS if the URB is submitted successfully. USBHST_INVALID_PARAMETER if the parameters are not valid. USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRhProcessStandardRequest()

NAME	usbEhcdRhProcessStandardRequest() – processes a standard transfer request
SYNOPSIS	<pre> USBHST_STATUS usbEhcdRhProcessStandardRequest (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ pUSBHST_URB pURB /* Ptr to User Request Block */) </pre>
DESCRIPTION	This routine processes a standard transfer request.
RETURNS	<p>USBHST_SUCCESS if the URB is submitted successfully.</p> <p>USBHST_INVALID_PARAMETER if the parameters are not valid.</p> <p>USBHST_INSUFFICIENT_BANDWIDTH if memory is insufficient for the request.</p>
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbEhcdRhSetPortFeature()

NAME	usbEhcdRhSetPortFeature() – set the features of the port
SYNOPSIS	<pre> USBHST_STATUS usbEhcdRhSetPortFeature (pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */ pUSBHST_URB pURB /* Ptr to User Request Block */) </pre>

DESCRIPTION	This routine sets the features of the port.
RETURNS	USBHST_SUCCESS if the URB is submitted successfully. USBHST_INVALID_PARAMETER if the parameters are not valid.
ERRNO	None.
SEE ALSO	usbEhcdRhEmulation

usbHalTcdAddressSet()

NAME	usbHalTcdAddressSet() – hal interface to set address.
SYNOPSIS	<pre>STATUS usbHalTcdAddressSet (pUSBHAL_TCD_NEXUS pNexus, /* TCD_NEXUS structure member */ UINT8 deviceAddress /* Address of the device to set */)</pre>
DESCRIPTION	This function sets an address on the target controller.
RETURNS	OK, if address set successfully; ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalDeviceControlStatus

usbHalTcdAttach()

NAME	usbHalTcdAttach() – attaches a TCD
SYNOPSIS	<pre>STATUS usbHalTcdAttach (USB_TCD_EXEC_FUNC tcdExecFunc, /* single entry point of TCD */ pVOID tcdParam, /* TCD specific paramter */ pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD */ pUSB_APPLN_DEVICE_INFO pDeviceInfo, /* USB_APPLN_DEVICE_INFO */)</pre>

```

        USB_TCD_MNGMT_CALLBACK mngmtCallback,          /* management callback
function */
        pVOID                  mngmtCallbackParam /* management callback
parameter */
    )

```

DESCRIPTION This sub-module attaches the Target Controller Driver.

RETURNS OK if TCD is attached successfully, **ERROR** otherwise.

ERRNO None.

SEE ALSO **usbHalInitExit**

usbHalTcdCurrentFrameGet()

NAME **usbHalTcdCurrentFrameGet()** – hal interface to get Current Frame Number.

SYNOPSIS

```

STATUS usbHalTcdCurrentFrameGet
(
    pUSBHAL_TCD_NEXUS pNexus,    /* USBHAL_TCD_NEXUS */
    pUINT16            pFrameNo /* Frame number */
)

```

DESCRIPTION This function gets the current frame number.

RETURNS OK if frame number is retrieved successfully, **ERROR** otherwise.

ERRNO None.

SEE ALSO **usbHalDeviceControlStatus**

usbHalTcdDetach()

NAME **usbHalTcdDetach()** – detaches a TCD

SYNOPSIS STATUS usbHalTcdDetach

```
(
    pUSBHAL_TCD_NEXUS pNexus    /* USBHAL_TCD_NEXUS */
)
```

DESCRIPTION	This usb-routine is used to detach the TCD. All active endpoints are deleted before the TCD is detached.
RETURNS	OK if TCD is detached successfully, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalInitExit

usbHalTcdDeviceFeatureClear()

NAME	usbHalTcdDeviceFeatureClear() – hal interface to clear feature on device.
SYNOPSIS	<pre>STATUS usbHalTcdDeviceFeatureClear (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ UINT16 uFeatureSelector /* Feature to be cleared */)</pre>
DESCRIPTION	This function clears a feature on the target controller.
RETURNS	OK if feature cleared successfully, ERROR otherwise.
ERRNO	none.
SEE ALSO	usbHalDeviceControlStatus

usbHalTcdDeviceFeatureSet()

NAME	usbHalTcdDeviceFeatureSet() – hal interface to set feature on the device.
SYNOPSIS	<pre>STATUS usbHalTcdDeviceFeatureSet (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */</pre>

```
UINT16      uFeatureSelector, /* Feature to set */
UINT8      uTestSelector     /* Test Mode arguments */
)
```

DESCRIPTION This function sets a feature on the target controller

RETURNS OK if feature set successfully, **ERROR** otherwise.

ERRNO None.

SEE ALSO **usbHalDeviceControlStatus**

usbHalTcdDisable()

NAME **usbHalTcdDisable()** – disables the target controller

SYNOPSIS

```
STATUS usbHalTcdDisable
(
    pUSBHAL_TCD_NEXUS pNexus /* USBHAL_TCD_NEXUS */
)
```

DESCRIPTION This sub-routine is used to disable the target controller.

RETURNS OK if target controller is successfully disabled, **ERROR** otherwise.

ERRNO None.

SEE ALSO **usbHalInitExit**

usbHalTcdEnable()

NAME **usbHalTcdEnable()** – enables the target controller.

SYNOPSIS

```
STATUS usbHalTcdEnable
(
    pUSBHAL_TCD_NEXUS pNexus /* USBHAL_TCD_NEXUS */
)
```

DESCRIPTION	This sub-routine is used to enable the Target Controller.
RETURNS	OK if target controller is successfully enabled, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalInitExit

usbHalTcdEndpointAssign()

NAME	usbHalTcdEndpointAssign() – configure an endpoint on the target controller
SYNOPSIS	<pre>STATUS usbHalTcdEndpointAssign (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ pUSB_ENDPOINT_DESCR pEndpointDesc, /* USB_ENDPOINT_DESCR */ UINT16 uConfigurationValue, /* configuration value */ UINT16 uInterface, /* interface number */ UINT16 uAltSetting, /* alternate setting */ pVOID * ppPipeHandle /* pointer to the Pipe handle */) */</pre>
DESCRIPTION	This function is used to configure an endpoint for USB operations. <i>pEndpointDesc</i> is the endpoint descriptor obtained from the above layer. On successfull configuration, we get a pipe handle <i>ppPipeHandle</i> which is used to carry out any further operations on that endpoint.
RETURNS	OK if endpoint is configured successfully, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalEndpoint

usbHalTcdEndpointRelease()

NAME	usbHalTcdEndpointRelease() – unconfigure endpoint on the target controller
------	----------------------------------------------------------------------------

SYNOPSIS	<pre> STATUS usbHalTcdEndpointRelease (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ pVOID pPipeHandle /* pipe handle */) </pre>
DESCRIPTION	This function is used to release an endpoint configured earlier. <i>pPipeHandle</i> is the handle to the pipe for the endpoint to be released.
RETURNS	OK if endpoint is unconfigured successfully, ERROR otherwise.
ERRNO	none.
SEE ALSO	usbHalEndpoint

usbHalTcdEndpointStateSet()

NAME	usbHalTcdEndpointStateSet() – set the state of an endpoint
SYNOPSIS	<pre> STATUS usbHalTcdEndpointStateSet (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ pVOID pPipeHandle, /* pipe handle */ UINT16 state /* state of the pipe */) </pre>
DESCRIPTION	This function is used to stall or un-stall an endpoint. <i>pPipeHandle</i> is the handle to the corresponding endpoint and <i>state</i> is the state to be set.
RETURNS	OK if endpoint state is set successfully, ERROR otherwise.
ERRNO	none.
SEE ALSO	usbHalEndpoint

usbHalTcdEndpointStatusGet()

NAME	usbHalTcdEndpointStatusGet() – get the status of an endpoint
-------------	----------------------------------------------------------------------

SYNOPSIS	<pre>STATUS usbHalTcdEndpointStatusGet (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ pVOID pPipeHandle, /* pipe handle */ pUINT8 pStatus /* pointer to hold the endpoint status */)</pre>
DESCRIPTION	This function is used to get the status of an endpoint. <i>pPipeHandle</i> is the handle to the corresponding endpoint and <i>pStatus</i> is the pointer to the status information obtained.
RETURNS	OK if endpoint status is retrieved successfully, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalEndpoint

usbHalTcdErpCancel()

NAME	usbHalTcdErpCancel() – cancel an ERP
SYNOPSIS	<pre>STATUS usbHalTcdErpCancel (pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */ pUSB_ERP pErp /* pointer to the ERP */)</pre>
DESCRIPTION	This sub-module is used to cancel the ERP submitted on an endpoint.
RETURNS	OK if ERP is cancelled successfully, ERROR otherwise.
ERRNO	none.
SEE ALSO	usbHalEndpoint

usbHalTcdErpSubmit()

NAME	usbHalTcdErpSubmit() – submit an ERP for an endpoint
SYNOPSIS	<pre>STATUS usbHalTcdErpSubmit</pre>


```
(
    pUSBHAL_TCD_NEXUS pNexus, /* USBHAL_TCD_NEXUS */
    pUSB_ERP          pErp    /* pointer to the ERP */
)
```

DESCRIPTION	This sub-module submits an ERP for transfer on an endpoint. The ERP structure consists of the pointer of the pipe-handle on which the ERP is submitted.
RETURNS	OK if ERP is submitted successfully, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalEndpoint

usbHalTcdSignalResume()

NAME	usbHalTcdSignalResume() – hal interface to initiate resume signal.
SYNOPSIS	<pre>STATUS usbHalTcdSignalResume (pUSBHAL_TCD_NEXUS pNexus /* USBHAL_TCD_NEXUS */)</pre>
DESCRIPTION	This function initiates resume signalling on the bus.
RETURNS	OK if resume signalling is initiated successfully, ERROR otherwise.
ERRNO	None.
SEE ALSO	usbHalDeviceControlStatus

usbHandleCreate()

NAME	usbHandleCreate() – Creates a new handle.
SYNOPSIS	<pre>STATUS usbHandleCreate (UINT32 handleSignature, /* Arbitrary handle signature */ </pre>

```
pVOID          handleParam,      /* Arbitrary handle parameter */
pGENERIC_HANDLE pHandle          /* Newly allocated handle */
)
```

DESCRIPTION	This routine creates a new handle. The caller passes an arbitrary <i>handleSignature</i> and a <i>handleParam</i> . The <i>handleSignature</i> will be used in subsequent calls to usbHandleValidate() .
RETURNS	OK or ERROR
ERRNO	S_usbHandleLib_NOT_INITIALIZED S_usbHandleLib_BAD_PARAM S_usbHandleLib_GENERAL_FAULT S_usbHandleLib_OUT_OF_HANDLES
SEE ALSO	usbHandleLib

usbHandleDestroy()

NAME	usbHandleDestroy() – Destroys a handle.
SYNOPSIS	STATUS usbHandleDestroy (GENERIC_HANDLE handle /* handle to be destroyed */)
DESCRIPTION	This routine destroys the <i>handle</i> created by calling usbHandleCreate() .
RETURNS	OK or ERROR
ERRNO	S_usbHandleLib_GENERAL_FAULT S_usbHandleLib_BAD_HANDLE
SEE ALSO	usbHandleLib

usbHandleInitialize()

NAME	usbHandleInitialize() – Initializes the handle utility library.
SYNOPSIS	STATUS usbHandleInitialize

```
(
UINT32 maxHandles /* max handles allocated by library */
)
```

DESCRIPTION	<p>This routine initializes the handle utility library. It must be called at least once before any other calls into the handle utility library. Calls to usbHandleInitialize() may be nested, allowing multiple clients to use the library without requiring that they be coordinated.</p> <p><i>maxHandles</i> defines the maximum number of handles which should be allocated by the library. Passing a zero in <i>maxHandles</i> causes the library to allocate a default number of handles. <i>maxHandles</i> is ignored on nested calls to usbHandleInitialize().</p>
RETURNS	OK or ERROR
ERRNO	<p>S_usbHandleLib_OUT_OF_MEMORY S_usbHandleLib_OUT_OF_RESOURCES</p>
SEE ALSO	usbHandleLib

usbHandleShutdown()

NAME	usbHandleShutdown() – Shuts down the handle utility library.
SYNOPSIS	STATUS usbHandleShutdown (void)
DESCRIPTION	<p>This routine shuts down the handle utility library. When calls to usbHandleInitialize() have been nested, usbHandleShutdown() must be called a corresponding number of times.</p>
RETURNS	OK or ERROR
ERRNO	None
SEE ALSO	usbHandleLib

usbHandleValidate()

NAME	usbHandleValidate() – Validates a handle.
-------------	--------------------------------------------------

SYNOPSIS	<pre>STATUS usbHandleValidate (GENERIC_HANDLE handle, /* handle to be validated */ UINT32 handleSignature, /* signature used to validate handle */ PVOID *pHandleParam /* Handle parameter on return */)</pre>
DESCRIPTION	This function validates <i>handle</i> . The <i>handle</i> must match the <i>handleSignature</i> used when the handle was originally created. If the handle is valid, the <i>pHandleParam</i> will be returned.
RETURNS	OK or ERROR
ERRNO	S_usbHandleLib_NOT_INITIALIZED S_usbHandleLib_BAD_HANDLE
SEE ALSO	usbHandleLib

usbHidIdleSet()

NAME	usbHidIdleSet() – Issues a SET_IDLE request to a USB HID.
SYNOPSIS	<pre>STATUS usbHidIdleSet (USBD_CLIENT_HANDLE usbClientHandle, /* caller's USBD client handle */ USBD_NODE_ID nodeId, /* desired node */ UINT16 interface, /* desired interface */ UINT16 reportId, /* desired report */ UINT16 duration /* idle duration */)</pre>
DESCRIPTION	Using the <i>usbClientHandle</i> provided by the caller, this function issues a SET_IDLE request to the indicated <i>nodeId</i> . The caller must also specify the <i>interface</i> , <i>reportId</i> , and <i>duration</i> . If the <i>duration</i> is zero, the idle period is infinite. If <i>duration</i> is non-zero, then it expresses time in four-msec units (for example, a <i>duration</i> of one = four msecs, two = eight msecs, and so forth). Refer to Section 7.2.4 of the USB HID specification for further details.
RETURNS	OK, or ERROR if unable to issue SET_IDLE request
ERRNO	None
SEE ALSO	usbLib

usbHidProtocolSet()

NAME	usbHidProtocolSet() – Issues a SET_PROTOCOL request to a USB HID.
SYNOPSIS	<pre> STATUS usbHidProtocolSet (USBD_CLIENT_HANDLE usbdClientHandle, /* caller's USBD client handle */ USBD_NODE_ID nodeId, /* desired node */ UINT16 interface, /* desired interface */ UINT16 protocol /* USB_HID_PROTOCOL_xxxx */) </pre>
DESCRIPTION	Using the <i>usbdClientHandle</i> provided by the caller, this routine issues a SET_PROTOCOL request to the indicated <i>nodeId</i> . The caller must specify the <i>interface</i> and the desired <i>protocol</i> . The <i>protocol</i> is expressed as USB_HID_PROTOCOL_xxxx . Refer to Section 7.2.6 of the USB HID specification for further details.
RETURNS	OK, or ERROR if unable to issue SET_PROTOCOL request
ERRNO	None
SEE ALSO	usbLib

usbHidReportSet()

NAME	usbHidReportSet() – Issues a SET_REPORT request to a USB HID.
SYNOPSIS	<pre> STATUS usbHidReportSet (USBD_CLIENT_HANDLE usbdClientHandle, /* caller's USBD client handle */ USBD_NODE_ID nodeId, /* desired node */ UINT16 interface, /* desired interface */ UINT16 reportType, /* report type */ UINT16 reportId, /* report Id */ pUINT8 reportBfr, /* report value */ UINT16 reportLen /* length of report */) </pre>
DESCRIPTION	Using the <i>usbdClientHandle</i> provided by the caller, this function issues a SET_REPORT request to the indicated <i>nodeId</i> . The caller must also specify the <i>interface</i> , <i>reportType</i> , <i>reportId</i> , <i>reportBfr</i> , and <i>reportLen</i> . Refer to Section 7.2.2 of the USB HID specification for further detail.
RETURNS	OK, or ERROR if unable to issue SET_REPORT request

ERRNO None

SEE ALSO **usbLib**

usbHstBusDeregister()

NAME **usbHstBusDeregister()** – deregister a USB Bus

SYNOPSIS USBHST_STATUS usbHstBusDeregister
 (
 UINT32 hHCDriver, /* Host Controller Driver handle */
 UINT32 uRelativeBusIndex, /* Bus index being deregistered */
 UINT32 hDefaultPipe /* Default pipe for USB bus */
)

DESCRIPTION This routine deregisters an USB Bus corresponding to the controller.

RETURNS **USBHST_SUCCESS**, **USBHST_INVALID_PARAMETER**,
USBHST_INSUFFICIENT_RESOURCES, **USBHST_FAILURE** when Attempt to deregister the
USB Bus while there are functional devices on it

ERRNO None

SEE ALSO **usb**d

usbHstBusRegister()

NAME **usbHstBusRegister()** – registers an USB Bus

SYNOPSIS USBHST_STATUS usbHstBusRegister
 (
 UINT32 hHCDriver, /* Host Controller Driver handle */
 UINT8 uSpeed, /* USB Bus speed */
 UINT32 hDefaultPipe, /* Default pipe handle */
 VXB_DEVICE_ID pDev /* struct vxbDev */
)

DESCRIPTION This routine registers an USB Bus corresponding to the host controller. This routine also
announces the host controller device to vxBus

RETURNS USBHST_SUCCESS, USBHST_INVALID_PARAMETER, USBHST_INSUFFICIENT_RESOURCES, USBHST_FAILURE if USB Bus is already registered

ERRNO None

SEE ALSO **usbd**

usbHstDriverDeregister()

NAME **usbHstDriverDeregister()** – deregisters USB class driver

SYNOPSIS

```
USBHST_STATUS  usbHstDriverDeregister
(
    pUSBHST_DEVICE_DRIVER pDeviceDriverInfo /* Ptr to Device Driver info */
)
```

DESCRIPTION This routine deregisters the class driver with the USB Stack.

RETURNS USBHST_INVALID_PARAMETER, USBHST_SUCCESS, USBHST_FAILURE if Driver is not found or if it is a hub class driver and there are some functional devices present

ERRNO None

SEE ALSO **usbd**

usbHstDriverRegister()

NAME **usbHstDriverRegister()** – register class driver

SYNOPSIS

```
USBHST_STATUS  usbHstDriverRegister
(
    pUSBHST_DEVICE_DRIVER  pDeviceDriverInfo, /* Ptr to Device Driver info */
    VOID                  ** pContext,         /* Ptr to context information */
    char                  * pDrvName           /* name of the driver */
)
```

DESCRIPTION This routine registers the class driver with the USB Host Stack. The function also register the class driver with vxBus.

RETURNS	USBHST_INVALID_PARAMETER, USBHST_SUCCESS, USBHST_FAILURE if Driver is already registered
ERRNO	None
SEE ALSO	usbd

usbHstHCDDeregister()

NAME	usbHstHCDDeregister() – deregister a Host Controller Driver
SYNOPSIS	<pre>USBHST_STATUS usbHstHCDDeregister (UINT32 hHCDriver /* Host Controller Driver handle */)</pre>
DESCRIPTION	This routine deregisters a Host Controller Driver with the USB Stack. This function also deregisters the host controller driver as bus controller from vxBus
RETURNS	USBHST_SUCCESS, USBHST_INVALID_PARAMETER, USBHST_FAILURE if bus count is not zero
ERRNO	None
SEE ALSO	usbd

usbHstHCDRegister()

NAME	usbHstHCDRegister() – register Host Controller Driver with USB
SYNOPSIS	<pre>USBHST_STATUS usbHstHCDRegister (pUSBHST_HC_DRIVER pHCDriver, /* Ptr to Host Controller driver */ UINT32 *phHCDriver, /* Ptr to Host Controller handle */ void * pContext, /* pContext information */ UINT32 busID /* bus Id */)</pre>
DESCRIPTION	This routine registers a Host Controller Driver with the USB Stack. The routine also registers the host controller driver as bus type with vxBus. This is done by calling the routine vxbBusTypeRegister (). This routine allocates memory for the structure

USBHST_HC_DRIVER :: **vxbBusTypeInfo** and populates the **busId** with host controller bus id "busID" which is passed as argument to the function.

RETURNS **USBHST_INVALID_PARAMETER**, **USBHST_INSUFFICIENT_RESOURCE**, **USBHST_SUCCESS** if Host Controller Driver is registered successfully

ERRNO None

SEE ALSO **usbd**

usbHubExit()

NAME **usbHubExit()** – de-registers and cleans up the USB Hub Class Driver.

SYNOPSIS `INT8 usbHubExit (void)`

DESCRIPTION de-registers and cleans up the USB Hub Class Driver from the USB Host Software Stack.

RETURNS None

ERRNO None

SEE ALSO **usbHubInitialization**

usbHubInit()

NAME **usbHubInit()** – registers USB Hub Class Driver function pointers.

SYNOPSIS `INT8 usbHubInit (void)`

DESCRIPTION This function initializes the global variables and registers USB Hub Class Driver function pointers with the USB Host Software Stack. This also retrieves the USB Host Software Stack functions for future access.

RETURNS 0 , -1 on fail.

ERRNO None

SEE ALSO **usbHubInitialization**

usbKeyboardDevInit()

NAME	usbKeyboardDevInit() – initialize USB keyboard SIO driver
SYNOPSIS	<code>STATUS usbKeyboardDevInit (void)</code>
DESCRIPTION	Initializes the USB keyboard SIO driver. The USB keyboard SIO driver maintains an initialization count, so calls to this function may be nested.
RETURNS	OK, or ERROR if unable to initialize.
ERRNO	S_usbKeyboardLib_OUT_OF_RESOURCES Sufficient resources are not available to create mutex S_usbKeyboardLib_USBD_FAULT Fault in the USB Layer
SEE ALSO	usbKeyboardLib

usbKeyboardDevShutdown()

NAME	usbKeyboardDevShutdown() – shuts down keyboard SIO driver
SYNOPSIS	<code>STATUS usbKeyboardDevShutdown (void)</code>
DESCRIPTION	This function shuts down the keyboard driver. The driver is shutdown only if <i>initCount</i> after decrementing. If it is more the 0, it is decremented.
RETURNS	OK, or ERROR if unable to shutdown.
ERRNO	S_usbKeyboardLib_NOT_INITIALIZED Keyboard Driver not initialized
SEE ALSO	usbKeyboardLib

usbKeyboardDynamicAttachRegister()

NAME	usbKeyboardDynamicAttachRegister() – Register keyboard attach callback
-------------	--------------------------------------------------------------------------------

SYNOPSIS	<pre> STATUS usbKeyboardDynamicAttachRegister (USB_KBD_ATTACH_CALLBACK callback, /* new callback to be registered */ pVOID arg /* user-defined arg to callback */) </pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre> typedef (*USB_KBD_ATTACH_CALLBACK) (pVOID arg, SIO_CHAN *pSioChan, UINT16 attachCode); </pre> <p>usbKeyboardLib will invoke <i>callback</i> each time a USB keyboard is attached to or removed from the system. <i>arg</i> is a caller-defined parameter which will be passed to the <i>callback</i> each time it is invoked. The <i>callback</i> will also be passed a pointer to the SIO_CHAN structure for the channel being created/destroyed and an attach code of USB_KBD_ATTACH or USB_KBD_REMOVE.</p>
RETURNS	OK, or ERROR if unable to register callback
ERRNO	<p>S_usbKeyboardLib_BAD_PARAM Bad Parameter are passed</p> <p>S_usbKeyboardLib_OUT_OF_MEMORY Not sufficient memory is available</p>
SEE ALSO	usbKeyboardLib

usbKeyboardDynamicAttachUnregister()

NAME	usbKeyboardDynamicAttachUnregister() – Unregisters keyboard attach callback
SYNOPSIS	<pre> STATUS usbKeyboardDynamicAttachUnRegister (USB_KBD_ATTACH_CALLBACK callback, /* callback to be unregistered */ pVOID arg /* user-defined arg to callback */) </pre>
DESCRIPTION	<p>This function cancels a previous request to be dynamically notified for keyboard attachment and removal. The <i>callback</i> and <i>arg</i> parameters must exactly match those passed in a previous call to usbKeyboardDynamicAttachRegister().</p>
RETURNS	OK, or ERROR if unable to unregister callback

ERRNO **S_usbKeyboardLib_NOT_REGISTERED**
 Could not register the callback

SEE ALSO **usbKeyboardLib**

usbKeyboardSioChanLock()

NAME **usbKeyboardSioChanLock()** – Marks **SIO_CHAN** structure as in use

SYNOPSIS

```
STATUS usbKeyboardSioChanLock
(
    SIO_CHAN *pChan /* SIO_CHAN to be marked as in use */
)
```

DESCRIPTION A caller uses **usbKeyboardSioChanLock()** to notify **usbKeyboardLib** that it is using the indicated **SIO_CHAN** structure. **usbKeyboardLib** maintains a count of callers using a particular **SIO_CHAN** structure so that it knows when it is safe to dispose of a structure when the underlying USB keyboard is removed from the system. So long as the "lock count" is greater than zero, **usbKeyboardLib** will not dispose of an **SIO_CHAN** structure.

RETURNS **OK**

ERRNO none.

SEE ALSO **usbKeyboardLib**

usbKeyboardSioChanUnlock()

NAME **usbKeyboardSioChanUnlock()** – Marks **SIO_CHAN** structure as unused

SYNOPSIS

```
STATUS usbKeyboardSioChanUnlock
(
    SIO_CHAN *pChan /* SIO_CHAN to be marked as unused */
)
```

DESCRIPTION This function releases a lock placed on an **SIO_CHAN** structure. When a caller no longer needs an **SIO_CHAN** structure for which it has previously called **usbKeyboardSioChanLock()**, then it should call this function to release the lock.

NOTE	If the underlying USB keyboard device has already been removed from the system, then this function will automatically dispose of the SIO_CHAN structure if this call removes the last lock on the structure. Therefore, a caller must not reference the SIO_CHAN again structure after making this call.
RETURNS	OK, or ERROR if unable to mark SIO_CHAN structure unused
ERRNO	S_usbKeyboardLib_NOT_LOCKED No lock to unlock
SEE ALSO	usbKeyboardLib

usbListFirst()

NAME	usbListFirst() – Returns first entry on a linked list.
SYNOPSIS	<pre>pVOID usbListFirst (pLIST_HEAD pListHead /* head of linked list */)</pre>
DESCRIPTION	This routine returns the pointer to the first structure in a linked list given a pointer to LIST_HEAD .
RETURNS	<i>pStruct</i> of first structure on list or NULL if list empty
ERRNO	None
SEE ALSO	usbListLib

usbListLink()

NAME	usbListLink() – Adds an element to a linked list.
SYNOPSIS	<pre>VOID usbListLink (pLIST_HEAD pHead, /* list head */ pVOID pStruct, /* ptr to base of structure to be linked */ pLINK pLink, /* ptr to LINK structure to be linked */)</pre>

```
        UINT16    flag        /* indicates LINK_HEAD or LINK_TAIL */
    )
```

DESCRIPTION	Using the link structure <i>pLink</i> , add <i>pStruct</i> to a list of which the list head is <i>pHead</i> . <i>flag</i> must be LINK_HEAD or LINK_TAIL .
RETURNS	N/A
ERRNO	None
SEE ALSO	usbListLib

usbListLinkProt()

NAME	usbListLinkProt() – Adds an element to a list guarded by a mutex.
SYNOPSIS	<pre>VOID usbListLinkProt (pLIST_HEAD pHead, /* list head */ pVOID pStruct, /* ptr to base of structure to be linked */ pLINK pLink, /* ptr to LINK structure to be linked */ UINT16 flag, /* indicates LINK_HEAD or LINK_TAIL */ MUTEX_HANDLE mutex /* list guard mutex */)</pre>
DESCRIPTION	This routine is similar to linkList() except that it will take the <i>mutex</i> before manipulating the list.
NOTE	The routine will block forever if the mutex does not become available.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbListLib

usbListNext()

NAME	usbListNext() – Retrieves the next pStruct in a linked list.
SYNOPSIS	<pre>pVOID usbListNext (pLINK pLink /* LINK structure */)</pre>
DESCRIPTION	This routine returns the pointer to the next structure in a linked list given a <i>pLink</i> pointer. The value returned is the <i>pStruct</i> of the element in the linked list which follows the current <i>pLink</i> , not a pointer to the following <i>pLink</i> . (Typically, a client is more interested in walking its own list of structures than in the link structures used to maintain the linked list.
RETURNS	<i>pStruct</i> of next structure in list or NULL if end of list.
ERRNO	None
SEE ALSO	usbListLib

usbListUnlink()

NAME	usbListUnlink() – Removes an entry from a linked list.
SYNOPSIS	<pre>VOID usbListUnlink (pLINK pLink /* LINK structure to be unlinked */)</pre>
DESCRIPTION	Removes <i>pLink</i> from a linked list.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbListLib

usbListUnlinkProt()

NAME	usbListUnlinkProt() – Removes an element from a list guarded by a mutex.
SYNOPSIS	<pre>VOID usbListUnlinkProt (pLINK pLink, /* LINK structure to be unlinked */ MUTEX_HANDLE mutex /* list guard mutex */)</pre>
DESCRIPTION	This routine is the same as usbListUnlink() except that it will take the <i>mutex</i> before manipulating the list.
NOTE	The function will block forever if the mutex does not become available.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbListLib

usbMouseDevInit()

NAME	usbMouseDevInit() – initialize USB mouse SIO driver
SYNOPSIS	<pre>STATUS usbMouseDevInit (void)</pre>
DESCRIPTION	Initializes the USB mouse SIO driver. The USB mouse SIO driver maintains an initialization count, so calls to this function may be nested.
RETURNS	OK, or ERROR if unable to initialize.
ERRNO	S_usbMouseLib_OUT_OF_RESOURCES Sufficient Resources are not available S_usbMouseLib_USBD_FAULT Error in USB Layer
SEE ALSO	usbMouseLib

usbMouseDevShutdown()

NAME	usbMouseDevShutdown() – shuts down mouse SIO driver
SYNOPSIS	<code>STATUS usbMouseDevShutdown (void)</code>
DESCRIPTION	This function shutdowns the mouse SIO driver. If after decrementing <i>initCount</i> is 0, SIO driver is uninitialized.
RETURNS	OK, or ERROR if unable to shutdown.
ERRNO	<code>S_usbMouseLib_NOT_INITIALIZED</code> SIO Driver is not initialized
SEE ALSO	usbMouseLib

usbMouseDynamicAttachRegister()

NAME	usbMouseDynamicAttachRegister() – Register mouse attach callback
SYNOPSIS	<pre>STATUS usbMouseDynamicAttachRegister (USB_MSE_ATTACH_CALLBACK callback, /* new callback to be registered */ pVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre>typedef (*USB_MSE_ATTACH_CALLBACK) (pVOID arg, SIO_CHAN *pSioChan, UINT16 attachCode);</pre> <p>usbMouseLib will invoke <i>callback</i> each time a USB mouse is attached to or removed from the system. <i>arg</i> is a caller-defined parameter which will be passed to the <i>callback</i> each time it is invoked. The <i>callback</i> will also be passed a pointer to the SIO_CHAN structure for the channel being created/destroyed and an attach code of USB_MSE_ATTACH or USB_MSE_REMOVE.</p>
RETURNS	OK, or ERROR if unable to register callback

ERRNO	S_usbMouseLib_BAD_PARAM Bad Parameter is passed
	S_usbMouseLib_OUT_OF_MEMORY Not sufficient memory available
SEE ALSO	usbMouseLib

usbMouseDynamicAttachUnregister()

NAME	usbMouseDynamicAttachUnregister() – Unregisters mouse attach callback
SYNOPSIS	<pre>STATUS usbMouseDynamicAttachUnRegister (USB_MSE_ATTACH_CALLBACK callback, /* callback to be unregistered */ PVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	This function cancels a previous request to be dynamically notified for mouse attachment and removal. The <i>callback</i> and <i>arg</i> paramters must exactly match those passed in a previous call to usbMouseDynamicAttachRegister() .
RETURNS	OK, or ERROR if unable to unregister callback
ERRNO	S_usbMouseLib_NOT_REGISTERED Could not register the callback
SEE ALSO	usbMouseLib

usbMouseSioChanLock()

NAME	usbMouseSioChanLock() – Marks SIO_CHAN structure as in use
SYNOPSIS	<pre>STATUS usbMouseSioChanLock (SIO_CHAN *pChan /* SIO_CHAN to be marked as in use */)</pre>
DESCRIPTION	A caller uses usbMouseSioChanLock() to notify usbMouseLib that it is using the indicated SIO_CHAN structure. usbMouseLib maintains a count of callers using a particular SIO_CHAN structure so that it knows when it is safe to dispose of a structure

when the underlying USB mouse is removed from the system. So long as the "lock count" is greater than zero, **usbMouseLib** will not dispose of an **SIO_CHAN** structure.

RETURNS **OK**

ERRNO none

SEE ALSO **usbMouseLib**

usbMouseSioChanUnlock()

NAME **usbMouseSioChanUnlock()** – Marks **SIO_CHAN** structure as unused

SYNOPSIS `STATUS usbMouseSioChanUnlock`
 (
 SIO_CHAN *pChan /* **SIO_CHAN** to be marked as unused */
)

DESCRIPTION This function releases a lock placed on an **SIO_CHAN** structure. When a caller no longer needs an **SIO_CHAN** structure for which it has previously called **usbMouseSioChanLock()**, then it should call this function to release the lock.

NOTE If the underlying USB mouse device has already been removed from the system, then this function will automatically dispose of the **SIO_CHAN** structure if this call removes the last lock on the structure. Therefore, a caller must not reference the **SIO_CHAN** again structure after making this call.

RETURNS **OK**, or **ERROR** if unable to mark **SIO_CHAN** structure unused

ERRNO **S_usbMouseLib_NOT_LOCKED**
 No lock to unlock

SEE ALSO **usbMouseLib**

usbMsBulkInErpInUseFlagGet()

NAME **usbMsBulkInErpInUseFlagGet()** – get the Bulk-in ERP inuse flag

SYNOPSIS `BOOL usbMsBulkInErpInUseFlagGet (void)`

DESCRIPTION	This function is used to get the state of the Bulk-In ERP.
RETURNS	TRUE or FALSE
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsBulkInErpInUseFlagSet()

NAME	usbMsBulkInErpInUseFlagSet() – set the Bulk-In ERP inuse flag
SYNOPSIS	<pre>void usbMsBulkInErpInUseFlagSet (BOOL state)</pre>
DESCRIPTION	This function is used to set the state of Bulk - IN ERP flag. <i>state</i> is the state to set.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsBulkInErpInit()

NAME	usbMsBulkInErpInit() – initialize the bulk-in ERP
SYNOPSIS	<pre>STATUS usbMsBulkInErpInit (UINT8 * pData, /* pointer to data */ UINT32 size, /* size of data */ ERP_CALLBACK erpCallback, /* erp callback */ PVOID usrPtr /* user pointer */)</pre>
DESCRIPTION	This function initializes the Bulk In ERP.

RETURNS OK, or **ERROR** if unable to submit ERP.

ERRNO none

SEE ALSO **usbTargMsLib**

usbMsBulkInStall()

NAME **usbMsBulkInStall()** – stall the bulk-in pipe

SYNOPSIS `STATUS usbMsBulkInStall (void)`

DESCRIPTION This routine stalls the bulk-in pipe.

RETURNS **OK** or **ERROR** if not able to stall the bulk IN endpoint.

ERRNO none

SEE ALSO **usbTargMsLib**

usbMsBulkInUnStall()

NAME **usbMsBulkInUnStall()** – unSTALL the bulk-in pipe

SYNOPSIS `STATUS usbMsBulkInUnStall (void)`

DESCRIPTION This routine unSTALLs the bulk-in pipe.

RETURNS **OK** or **ERROR** if not able to un-stall the bulk IN endpoint.

ERRNO none

SEE ALSO **usbTargMsLib**

usbMsBulkOutErpInUseFlagGet()

NAME	usbMsBulkOutErpInUseFlagGet() – get the Bulk-Out ERP inuse flag
SYNOPSIS	<code>BOOL usbMsBulkOutErpInUseFlagGet (void)</code>
DESCRIPTION	This function is used to get the state of the Bulk-OUT ERP.
RETURNS	OK, or ERROR if unable to submit ERP.
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsBulkOutErpInUseFlagSet()

NAME	usbMsBulkOutErpInUseFlagSet() – set the Bulk-Out ERP inuse flag
SYNOPSIS	<pre>void usbMsBulkOutErpInUseFlagSet (BOOL state /* State to set */)</pre>
DESCRIPTION	This function is used to set the state of Bulk - OUT ERP flag. <i>state</i> is the state to set.
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsBulkOutErpInit()

NAME	usbMsBulkOutErpInit() – initialize the bulk-Out ERP
SYNOPSIS	<code>STATUS usbMsBulkOutErpInit</code>

```
(
  UINT8      * pData,      /* pointer to buffer */
  UINT32     size,        /* size of data */
  ERP_CALLBACK erpCallback, /* IRP_CALLBACK */
  pVOID      usrPtr       /* user pointer */
)
```

DESCRIPTION This function initializes the bulk Out ERP.

RETURNS OK, or ERROR if unable to submit ERP.

ERRNO N/A

SEE ALSO **usbTargMsLib**

usbMsBulkOutStall()

NAME **usbMsBulkOutStall()** – stall the bulk-out pipe

SYNOPSIS STATUS usbMsBulkOutStall (void)

DESCRIPTION This routine stalls the bulk-out pipe.

RETURNS OK or ERROR in unable to stall the bulk OUT endpoints.

ERRNO none.

SEE ALSO **usbTargMsLib**

usbMsBulkOutUnStall()

NAME **usbMsBulkOutUnStall()** – unSTALL the bulk-out pipe

SYNOPSIS STATUS usbMsBulkOutUnStall (void)

DESCRIPTION This routine uninstalls the bulk-out pipe.

RETURNS OK or ERROR if not able to unSTALL the bulk out endpoints

ERRNO none

SEE ALSO **usbTargMsLib**

usbMsCBWGet()

NAME **usbMsCBWGet()** – get the last mass storage CBW received

SYNOPSIS `USB_BULK_CBW *usbMsCBWGet (void)`

DESCRIPTION This routine retrieves the last CBW received on the bulk-out pipe.

RETURNS **USB_BULK_CBW**

ERRNO none.

SEE ALSO **usbTargMsLib**

usbMsCBWInit()

NAME **usbMsCBWInit()** – initialize the mass storage CBW

SYNOPSIS `USB_BULK_CBW *usbMsCBWInit (void)`

DESCRIPTION This routine initializes the CBW by resetting all fields to their default value.

RETURNS **USB_BULK_CBW**

ERRNO none.

SEE ALSO **usbTargMsLib**

usbMsCSWGet()

NAME	usbMsCSWGet() – get the current CSW
SYNOPSIS	USB_BULK_CSW *usbMsCSWGet (void)
DESCRIPTION	This routine retrieves the current CSW.
RETURNS	USB_BULK_CSW
ERRNO	none.
SEE ALSO	usbTargMsLib

usbMsCSWInit()

NAME	usbMsCSWInit() – initialize the CSW
SYNOPSIS	USB_BULK_CSW *usbMsCSWInit (void)
DESCRIPTION	This routine initializes the CSW.
RETURNS	USB_BULK_CSW
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsisConfigured()

NAME	usbMsisConfigured() – test if the device is configured
SYNOPSIS	BOOL usbMsisConfigured (void)
DESCRIPTION	This function checks whether the device is configured or not.

RETURNS	TRUE or FALSE
ERRNO	none
SEE ALSO	usbTargMsLib

usbMsTestRxCallback()

NAME	usbMsTestRxCallback() – invoked after test data is received
SYNOPSIS	<pre>void usbMsTestRxCallback (pVOID p)</pre>
DESCRIPTION	This function is invoked after the Bulk OUT test data is transmitted. It sets the bulk OUT flag to <i>false</i> .
RETURNS	N/A
ERRNO	N/A
SEE ALSO	usbTargMsLib

usbMsTestTxCallback()

NAME	usbMsTestTxCallback() – invoked after test data transmitted
SYNOPSIS	<pre>void usbMsTestTxCallback (pVOID p)</pre>
DESCRIPTION	This function is invoked after the Bulk IN test data is transmitted. It sets the bulk IN flag to <i>false</i> .
RETURNS	N/A

ERRNO none

SEE ALSO `usbTargMsLib`

usbOhcdInit()

NAME `usbOhcdInit()` – initialize the USB OHCI Host Controller Driver.

SYNOPSIS `VOID usbOhcdInit (void)`

DESCRIPTION This function initializes internal data structures in the OHCI Host Controller Driver. This routine is typically called prior the the `vxBus` invocation of the device connect.

This routine requires that the USB D has been initialized.

This function registers the OHCI HCD with the USB D Layer.

PARAMETERS None

RETURNS `TRUE` if the OHCI Host Controllers are initialized, otherwise `FALSE`

ERRNO None.

SEE ALSO `usbOhci`

usbOhciDumpEndpointDescriptor()

NAME `usbOhciDumpEndpointDescriptor()` – dump endpoint descriptor contents

SYNOPSIS `VOID usbOhciDumpEndpointDescriptor
(
 PVOID pEndpointDescriptor
)`

DESCRIPTION This function is used to dump the contents of the endpoint descriptor.

PARAMETERS *pEndpointDescriptor (IN)* - Pointer to the endpoint descriptor to be dumped.

RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciDumpGeneralTransferDescriptor()

NAME	usbOhciDumpGeneralTransferDescriptor() – dump general transfer descriptor
SYNOPSIS	<pre>VOID usbOhciDumpGeneralTransferDescriptor (PVOID pGeneralTransferDescriptor)</pre>
DESCRIPTION	This function is used to dump the contents of the general transfer descriptor.
PARAMETERS	<i>pGeneralTransferDescriptor (IN)</i> - Pointer to the general descriptor
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciDumpMemory()

NAME	usbOhciDumpMemory() – dump memory contents
SYNOPSIS	<pre>VOID usbOhciDumpMemory (UINT32 uAddress, UINT32 uLength, UINT32 uWidth)</pre>
DESCRIPTION	This function is used to dump the contents of the specified memory location.

PARAMETERS	<i>uAddress (IN)</i> - Specifies the address of memory location. <i>uLength (IN)</i> - Specifies the length of memory to be dumped. <i>uWidth (IN)</i> - Specifies the width of each entry in bytes. For example, if this value is 1, the data will be displayed in bytes. If the value is 4, the data will be displayed in DWORDS (4 bytes).
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciDumpPendingTransfers()

NAME	usbOhciDumpPendingTransfers() – dump pending transfers
SYNOPSIS	<pre>VOID usbOhciDumpPendingTransfers (PVOID pEndpointDescriptor)</pre>
DESCRIPTION	This function is used to dump the pending transfers for the endpoint
PARAMETERS	<i>pEndpointDescriptor (IN)</i> - Pointer to the endpoint descriptor to be dumped
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciDumpPeriodicEndpointList()

NAME	usbOhciDumpPeriodicEndpointList() – dump periodic endpoint descriptor list
SYNOPSIS	<pre>VOID usbOhciDumpPeriodicEndpointList</pre>

```
(  
    UINT8 uHostControllerIndex  
)
```

DESCRIPTION	This function is used to dump the contents of the periodic endpoint descriptor list.
PARAMETERS	<i>uHostControllerIndex</i> (IN) - Specifies the host controller index
RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciDumpRegisters()

NAME	usbOhciDumpRegisters() – dump registers contents.
SYNOPSIS	<pre>BOOLEAN usbOhciDumpRegisters((UINT32 uHostControllerIndex)</pre>
DESCRIPTION	This function is used to dump the contents of the USB OHCI Host Controller Registers.
PARAMETERS	<i>uHostControllerIndex</i> (IN) - Specifies the OHCI Host Controller index.
RETURNS	TRUE if the host controller index specified is valid for the USB OHCI controllers detected on the system, otherwise FALSE .
ERRNO	None.
SEE ALSO	usbOhciDebug

usbOhciExit()

NAME	usbOhciExit() – uninitialize the USB OHCI Host Controller Driver.
------	---------------------------------------------------------------------------

SYNOPSIS	<code>BOOLEAN usbOhciExit (void)</code>
DESCRIPTION	This function uninitializes the OHCI Host Controller Driver.
RETURNS	<code>FALSE</code> , <code>TRUE</code> if all the OHCI Host Controllers are reset and the cleanup is successful.
ERRNO	None.
SEE ALSO	<code>usbOhci</code>

usbOhciInitializeModuleTestingFunctions()

NAME	<code>usbOhciInitializeModuleTestingFunctions()</code> – obtains entry points
SYNOPSIS	<pre> VOID usbOhciInitializeModuleTestingFunctions (PUSBHST_HC_DRIVER_TEST pHCDriverTestEntryPoints /* Ptr to HCD module entry points */) </pre>
DESCRIPTION	Function to obtain the entry points used for HCD module testing.
RETURNS	N/A
ERRNO	none
SEE ALSO	<code>usbOhciDebug</code>

usbOhciInstantiate()

NAME	<code>usbOhciInstantiate()</code> – instantiate the USB OHCI Host Controller Driver.
SYNOPSIS	<code>VOID usbOhciInstantiate (void)</code>
DESCRIPTION	This routine instantiates the OHCI Host Controller Driver and allows the OHCI Controller driver to be included with the vxWorks image and not be registered with vxBus. OHCI devices will remain orphan devices until the <code>usbOhciInit()</code> routine is called. This supports the <code>INCLUDE_OHCI</code> behaviour of previous vxWorks releases.

The routine itself does nothing.

RETURNS	N/A
ERRNO	None.
SEE ALSO	usbOhci

usbPegasusDevLock()

NAME **usbPegasusDevLock()** – marks USB_PEGASUS_DEV structure as in use

SYNOPSIS

```
STATUS usbPegasusDevLock
(
    USBD_NODE_ID nodeId /* NodeId of the USB_PEGASUS_DEV */
                        /* to be marked as in use          */
)
```

DESCRIPTION A caller uses **usbPegasusDevLock()** to notify **usbPegasusDevLib** that it is using the indicated PEGASUS device structure. **usbPegasusDevLib** maintains a count of callers using a particular Pegasus Device structure so that it knows when it is safe to dispose of a structure when the underlying Pegasus Device is removed from the system. So long as the "lock count" is greater than zero, **usbPegasusDevLib** will not dispose of an Pegasus structure.

RETURNS OK, or ERROR if unable to mark Pegasus structure in use.

ERRNO none

SEE ALSO **usbPegasusEnd**

usbPegasusDevUnlock()

NAME **usbPegasusDevUnlock()** – marks USB_PEGASUS_DEV structure as unused

SYNOPSIS

```
STATUS usbPegasusDevUnlock
(
    USBD_NODE_ID nodeId /* NodeId of the BLK_DEV to be marked as unused */
)
```


DESCRIPTION	This function releases a lock placed on an Pegasus Device structure. When a caller no longer needs an Pegasus Device structure for which it has previously called usbPegasusDevLock() , then it should call this function to release the lock.
NOTE	If the underlying Pegasus device has already been removed from the system, then this function will automatically dispose of the Pegasus Device structure if this call removes the last lock on the structure. Therefore, a caller must not reference the Pegasus Device structure after making this call.
RETURNS	OK, or ERROR if unable to mark Pegasus Device structure unused.
ERRNO	S_usbPegasusLib_NOT_LOCKED No lock to Unlock
SEE ALSO	usbPegasusEnd

usbPegasusDynamicAttachRegister()

NAME	usbPegasusDynamicAttachRegister() – register PEGASUS device attach callback
SYNOPSIS	<pre>STATUS usbPegasusDynamicAttachRegister (USB_PEGASUS_ATTACH_CALLBACK callback, /* new callback to be registered */ pVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre>typedef (*USB_PEGASUS_ATTACH_CALLBACK) (pVOID arg, USB_PEGASUS_DEV * pDev, UINT16 attachCode);</pre> <p>usbPegasusDevLib will invoke <i>callback</i> each time a PEGASUS device is attached to or removed from the system. <i>arg</i> is a caller-defined parameter which will be passed to the <i>callback</i> each time it is invoked. The <i>callback</i> will also pass the structure of the device being created/destroyed and an attach code of USB_PEGASUS_ATTACH or USB_PEGASUS_REMOVE.</p>
NOTE	The user callback routine should not invoke any driver function that submits IRPs. Further processing must be done from a different task context. As the driver routines wait for IRP completion, they cannot be invoked from USB client task's context created for this driver.

RETURNS	OK, or ERROR if unable to register callback
ERRNO	S_usbPegasusLib_BAD_PARAM Bad Parameter received S_usbPegasusLib_OUT_OF_MEMORY Sufficient memory no available
SEE ALSO	usbPegasusEnd

usbPegasusDynamicAttachUnregister()

NAME	usbPegasusDynamicAttachUnregister() – unregisters PEGASUS attach callbackx
SYNOPSIS	<pre>STATUS usbPegasusDynamicAttachUnregister (USB_PEGASUS_ATTACH_CALLBACK callback, /* callback to be unregistered */ pVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	This function cancels a previous request to be dynamically notified for attachment and removal. The <i>callback</i> and <i>arg</i> paramters must exactly match those passed in a previous call to usbPegasusDynamicAttachRegister() .
RETURNS	OK, or ERROR if unable to unregister the callback.
ERRNO	S_usbPegasusLib_NOT_REGISTERED Could not regsiter the attachment callback
SEE ALSO	usbPegasusEnd

usbPegasusEndInit()

NAME	usbPegasusEndInit() – initializes the pegasus library
SYNOPSIS	<pre>STATUS usbPegasusEndInit(void)</pre>
DESCRIPTION	Initizes the pegasus library. The library maintains an initialization count so that the calls to this function might be nested.

This function initializes the system resources required for the library initializes the linked list for the ethernet devices found. This function reregisters the library as a client for the usbd calls and registers for dynamic attachment notification of usb communication device class and Ethernet sub class of devices.

RETURNS	OK if successful, ERROR if failure
ERRNO	S_usbPegasusLib_OUT_OF_RESOURCES Sufficient Resources not Available S_usbPegasusLib_USBD_FAULT Fault in the USBD Layer
SEE ALSO	usbPegasusEnd

usbPegasusEndLoad()

NAME usbPegasusEndLoad() – initialize the driver and device

SYNOPSIS

```
END_OBJ * usbPegasusEndLoad
(
    char * initString /* initialization string */
)
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

This function first extracts the currently attached pegasus device nodeId from the initialization string using the **pegasusEndParse()** function. It then passes these parameters and its control structure to the **pegasusDevInit()** function. **pegasusDevInit()** does most of the device specific initialization and brings the device to the operational state. Please refer to **pegasusLib.c** for more details about **usbenetDevInit()**. This driver will be attached to MUX and then the memory initialization of the device is carried out using **pegasusEndMemInit()**.

This function doesn't do any thing device specific. Instead, it delegates such initialization to **pegasusDevInit()**. This routine handles the other part of the driver initialization as required by MUX.

muxDevLoad calls this function twice. First time this function is called, initialization string will be NULL. We are required to fill in the device name ("usb") in the string and return. The next time this function is called the initialization string will be proper.

initString will be in the following format : "unit:nodeId:noOfInBfrs:noOfIrps"

PARAMETERS	<i>initString</i> The device initialization string.
RETURNS	An END object pointer or NULL on error.
ERRNO	none
SEE ALSO	usbPegasusEnd

usbPegasusEndUninit()

NAME	usbPegasusEndUninit() – un-initializes the pegasus class driver
SYNOPSIS	STATUS usbPegasusEndUninit (void)
DESCRIPTION	This function un-initializes the Pegasus Class Driver. It releases all the occupied resources. Everytime the function is called the global <i>initCount</i> will be decremented. The driver will be truly un-initialized only when <i>initCount</i> is 0.
RETURNS	OK if successful, ERROR if failure
ERRNO	none
SEE ALSO	usbPegasusEnd

usbPegasusReadReg()

NAME	usbPegasusReadReg() – read contents of specified and print
SYNOPSIS	STATUS usbPegasusReadReg (USBD_NODE_ID devId, /* pointer to device */ UINT8 offSet, /* Offset of the registers */ UINT8 noOfRegs /* No of registers to be read */)
DESCRIPTION	This function reads the register contents of Pegasus and prints them for debugging purposes.
RETURNS	OK if successful or ERROR on failure

ERRNO none

SEE ALSO **usbPegasusEnd**

usbPrinterDevInit()

NAME **usbPrinterDevInit()** – initialize USB printer SIO driver

SYNOPSIS `STATUS usbPrinterDevInit (void)`

DESCRIPTION Initializes the USB printer SIO driver. The USB printer SIO driver maintains an initialization count, so calls to this function may be nested.

RETURNS OK, or ERROR if unable to initialize.

ERRNO **S_usbPrinterLib_OUT_OF_RESOURCES**
Sufficient resources not available

S_usbPrinterLib_USBD_FAULT
Error in USBD layer

SEE ALSO **usbPrinterLib**

usbPrinterDevShutdown()

NAME **usbPrinterDevShutdown()** – shuts down printer SIO driver

SYNOPSIS `STATUS usbPrinterDevShutdown (void)`

DESCRIPTION This function shutdowns the printer SIO driver when *initCount* becomes 0

RETURNS OK, or ERROR if unable to shutdown.

ERRNO **S_usbPrinterLib_NOT_INITIALIZED**
Printer not initialized

SEE ALSO **usbPrinterLib**

usbPrinterDynamicAttachRegister()

NAME	usbPrinterDynamicAttachRegister() – Register printer attach callback
SYNOPSIS	<pre>STATUS usbPrinterDynamicAttachRegister (USB_PRN_ATTACH_CALLBACK callback, /* new callback to be registered */ pVOID arg /* user-defined arg to callback */)</pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre>typedef (*USB_PRN_ATTACH_CALLBACK) (pVOID arg, SIO_CHAN *pSioChan, UINT16 attachCode);</pre> <p>usbPrinterLib will invoke <i>callback</i> each time a USB printer is attached to or removed from the system. <i>arg</i> is a caller-defined parameter which will be passed to the <i>callback</i> each time it is invoked. The <i>callback</i> will also be passed a pointer to the SIO_CHAN structure for the channel being created/destroyed and an attach code of USB_PRN_ATTACH or USB_PRN_REMOVE.</p>
RETURNS	OK, or ERROR if unable to register callback
ERRNO	<p>S_usbPrinterLib_BAD_PARAM Bad Parameters received</p> <p>S_usbPrinterLib_OUT_OF_MEMORY System out of memory</p>
SEE ALSO	usbPrinterLib

usbPrinterDynamicAttachUnregister()

NAME	usbPrinterDynamicAttachUnregister() – Unregisters printer attach callback
SYNOPSIS	<pre>STATUS usbPrinterDynamicAttachUnRegister (USB_PRN_ATTACH_CALLBACK callback, /* callback to be unregistered */ pVOID arg /* user-defined arg to callback */)</pre>

DESCRIPTION	This function cancels a previous request to be dynamically notified for printer attachment and removal. The <i>callback</i> and <i>arg</i> paramters must exactly match those passed in a previous call to usbPrinterDynamicAttachRegister() .
RETURNS	OK, or ERROR if unable to unregister callback
ERRNO	S_usbPrinterLib_NOT_REGISTERED Could not register the attachment callback
SEE ALSO	usbPrinterLib

usbPrinterSioChanLock()

NAME	usbPrinterSioChanLock() – Marks SIO_CHAN structure as in use
SYNOPSIS	<pre>STATUS usbPrinterSioChanLock (SIO_CHAN *pChan /* SIO_CHAN to be marked as in use */)</pre>
DESCRIPTION	A caller uses usbPrinterSioChanLock() to notify usbPrinterLib that it is using the indicated SIO_CHAN structure. usbPrinterLib maintains a count of callers using a particular SIO_CHAN structure so that it knows when it is safe to dispose of a structure when the underlying USB printer is removed from the system. So long as the "lock count" is greater than zero, usbPrinterLib will not dispose of an SIO_CHAN structure.
RETURNS	OK, or ERROR if unable to mark SIO_CHAN structure in use.
ERRNO	none
SEE ALSO	usbPrinterLib

usbPrinterSioChanUnlock()

NAME	usbPrinterSioChanUnlock() – Marks SIO_CHAN structure as unused
SYNOPSIS	<pre>STATUS usbPrinterSioChanUnlock (SIO_CHAN *pChan /* SIO_CHAN to be marked as unused */)</pre>

DESCRIPTION	This function releases a lock placed on an SIO_CHAN structure. When a caller no longer needs an SIO_CHAN structure for which it has previously called usbPrinterSioChanLock() , then it should call this function to release the lock.
NOTE	If the underlying USB printer device has already been removed from the system, then this function will automatically dispose of the SIO_CHAN structure if this call removes the last lock on the structure. Therefore, a caller must not reference the SIO_CHAN again structure after making this call.
RETURNS	OK , or ERROR if unable to mark SIO_CHAN structure unused
ERRNO	S_usbPrinterLib_NOT_LOCKED No lock to unlock
SEE ALSO	usbPrinterLib

usbQueueCreate()

NAME	usbQueueCreate() – Creates an OS-independent queue structure.
SYNOPSIS	<pre>STATUS usbQueueCreate (UINT16 depth, /* Max entries queue can handle */ pQUEUE_HANDLE pQueueHandle /* Handle of newly created queue */)</pre>
DESCRIPTION	This routine creates a queue which can accomodate a number of USB_MESSAGE entries according to the <i>depth</i> parameter. It returns the <pQueueHandle) of the newly created queue if successful.
RETURNS	OK or ERROR
ERRNO	S_usbQueueLib_BAD_PARAMETER S_usbQueueLib_OUT_OF_MEMORY S_usbQueueLib_OUT_OF_RESOURCES
SEE ALSO	usbQueueLib

usbQueueDestroy()

NAME	usbQueueDestroy() – Destroys a queue.
SYNOPSIS	<pre>STATUS usbQueueDestroy (QUEUE_HANDLE queueHandle /* Handle of queue to destroy */)</pre>
DESCRIPTION	This function destroys a queue created by calling usbQueueCreate() .
RETURNS	OK or ERROR
ERRNO	S_usbQueueLib_BAD_HANDLE
SEE ALSO	usbQueueLib

usbQueueGet()

NAME	usbQueueGet() – Retrieves a message from a queue.
SYNOPSIS	<pre>STATUS usbQueueGet (QUEUE_HANDLE queueHandle, /* queue handle */ pUSB_MESSAGE pMsg, /* USB_MESSAGE to receive msg */ UINT32 blockFlag /* specifies blocking action */)</pre>
DESCRIPTION	This routine retrieves a message from the specified <i>queueHandle</i> and stores it in <i>pOssMsg</i> . If the queue is empty, <i>blockFlag</i> specifies the blocking behavior. OSS_BLOCK blocks indefinitely. OSS_DONT_BLOCK does not block and returns an error immediately if the queue is empty. Other values of <i>blockFlag</i> are interpreted as a count of milliseconds to block before declaring a failure.
RETURNS	OK or ERROR
ERRNO	S_usbQueueLib_BAD_HANDLE S_usbQueueLib_Q_NOT_AVAILABLE
SEE ALSO	usbQueueLib

usbQueuePut()

NAME **usbQueuePut()** – Puts a message into a queue.

SYNOPSIS

```
STATUS usbQueuePut
(
    QUEUE_HANDLE queueHandle, /* queue handle */
    UINT16      msg,          /* app-specific message */
    UINT16      wParam,       /* app-specific parameter */
    UINT32      lParam,       /* app-specific parameter */
    UINT32      blockFlag     /* specifies blocking action */
)
```

DESCRIPTION Places the specified *msg*, *wParam*, and *lParam* into *queueHandle*. This function will block only if the specified queue is full. If the queue is full, *blockFlag* specifies the blocking behavior. **OSS_BLOCK** blocks indefinitely. **OSS_DONT_BLOCK** does not block and returns an error if the queue is full. Other values of *blockFlag* are interpreted as a count of milliseconds to block before declaring a failure.

RETURNS OK or ERROR

ERRNO **S_usbQueueLib_BAD_HANDLE**
S_usbQueueLib_Q_NOT_AVAILABLE

SEE ALSO **usbQueueLib**

usbRecurringTime()

NAME **usbRecurringTime()** – calculates recurring time for interrupt/isoeh transfers.

SYNOPSIS

```
UINT32 usbRecurringTime
(
    UINT16 transferType, /* transfer type */
    UINT16 direction,    /* transfer direction */
    UINT16 speed,        /* speed of pipe */
    UINT16 packetSize,   /* max packet size for endpoint */
    UINT32 bandwidth,    /* bytes/frame or bytes/sec depending on pipe */
    UINT32 hostDelay,     /* host controller delay per packet */
    UINT32 hostHubLsSetup /* host controller time for low-speed setup */
)
```

DESCRIPTION For recurring transfers (for example, interrupt or isochronous transfers) an HCD needs to be able to calculate the bus time, measured in nanoseconds, which will be used by the transfer.

transferType specifies the type of transfer. For **USB_XFRTYPE_CONTROL** and **USB_XFRTYPE_BULK**, the calculated time is always 0. These are not recurring transfers. For **USB_XFRTYPE_INTERRUPT**, *bandwidth* must express the number of bytes to be transferred in each frame. For **USB_XFRTYPE_ISOCH**, *bandwidth* must express the number of bytes to be transferred in each second. This parameter is treated differently to allow greater flexibility in determining the true bandwidth requirements for each type of pipe.

RETURNS worst case number of nanoseconds required for transfer

ERRNO None

SEE ALSO **usbLib**

usbRegRead16()

NAME **usbRegRead16()** – reads 16-bit USB Register Space

SYNOPSIS UINT16 usbRegRead16
 (
 struct vxbDev * pDev, /* struct vxbDev * */
 void * pRegBase, /* pointer to base address */
 UINT32 offset, /* offset value */
 UINT32 flags /* vxBus access routine flag */
)

DESCRIPTION This routine will be used to read the 16 bit register value. The routine will call the vxBus provided access routine to carry out 8-bit read operation.

RETURNS 16 bit value read from register space

ERRNO none

SEE ALSO **usbVxbRegAccess**

usbRegRead32()

NAME **usbRegRead32()** – reads 32-bit USB Register Space

SYNOPSIS UINT32 usbRegRead32

usbRegRead8()

```

(
  struct vxbDev      * pDev,          /* struct vxbDev * */
  void               * pRegBase,      /* pointer to base address */
  UINT32             offset,          /* offset value */
  UINT32             flags            /* vxBus access routine flag */
)

```

DESCRIPTION	This routine will be used to read the 32 bit register value. The routine uses vxDev::vxbAccessList to read the register. The argument of this routine consist of pointer to struct vxbDev type, pointer to base address and offset that should be added to base address.
RETURNS	32 bit value read from register space
ERRNO	none
SEE ALSO	usbVxbRegAccess

usbRegRead8()

NAME **usbRegRead8()** – reads 8-bit USB Register Space

SYNOPSIS

```

UINT8 usbRegRead8
(
  struct vxbDev      * pDev,          /* struct vxbDev * */
  void               * pRegBase,      /* pointer to base address */
  UINT32             offset,          /* offset value */
  UINT32             flags            /* vxBus access routine flag */
)

```

DESCRIPTION	This routine will be used to read the 8 bit register value. The routine will call the vxBus provided access routine to carry out 8-bit read operation.
RETURNS	8 bit value read from register space
ERRNO	none
SEE ALSO	usbVxbRegAccess

usbRegWrite16()

NAME **usbRegWrite16()** – writes into 16-bit USB Register Space

SYNOPSIS

```

VOID usbRegWrite16
(
    struct vxbDev * pDev,      /* struct vxbDev * */
    void * pRegBase,          /* pointer to base address */
    UINT32 offset,            /* offset value */
    UINT16 value,              /* value to write */
    UINT32 flags               /* vxBus access routine flag */
)

```

DESCRIPTION This routine will be used to write to the 16-bit register value. The routine uses vxDev::vxbAccessList to write the register. The argument of this routine consist of pointer to struct vxbDev type, pointer to base address and offset that should be added to base address. *value* field consist of the value to write

RETURNS N/A

ERRNO none

SEE ALSO usbVxbRegAccess

usbRegWrite32()

NAME usbRegWrite32() – writes into 32-bit USB Register Space

SYNOPSIS

```

VOID usbRegWrite32
(
    struct vxbDev * pDev,      /* struct vxbDev * */
    void * pRegBase,          /* pointer to base address */
    UINT32 offset,            /* offset value */
    UINT32 value,              /* value to write */
    UINT32 flags               /* vxBus access routine flag */
)

```

DESCRIPTION This routine will be used to write to the 32 bit register value. The routine uses vxDev::vxbAccessList to write the register. The argument of this routine consist of pointer to struct vxbDev type, pointer to base address and offset that should be added to base address. *value* field consist of the value to write

RETURNS N/A

ERRNO none

SEE ALSO usbVxbRegAccess

usbSpeakerDevInit()

NAME	usbSpeakerDevInit() – initialize USB speaker SIO driver
SYNOPSIS	STATUS usbSpeakerDevInit (void)
DESCRIPTION	Initializes the USB speaker SIO driver. The USB speaker SIO driver maintains an initialization count, so calls to this function may be nested.
RETURNS	OK, or ERROR if unable to initialize.
ERRNO	S_usbSpeakerLib_OUT_OF_RESOURCES Sufficient resources not available S_usbSpeakerLib_USBD_FAULT Fault in the USBD Layer
SEE ALSO	usbSpeakerLib

usbSpeakerDevShutdown()

NAME	usbSpeakerDevShutdown() – shuts down speaker SIO driver
SYNOPSIS	STATUS usbSpeakerDevShutdown (void)
DESCRIPTION	This function shuts down speaker SIO driver depending on <i>initCount</i> . Every call to this function decrements the <i>initCount</i> , and when it turns 0, SIO speaker driver is shutdown.
RETURNS	OK, or ERROR if unable to shutdown.
ERRNO	S_usbSpeakerLib_NOT_INITIALIZED Speaker SIO Driver is not initialized
SEE ALSO	usbSpeakerLib

usbSpeakerDynamicAttachRegister()

NAME	usbSpeakerDynamicAttachRegister() – Register speaker attach callback
------	-----------------------------------------------------------------------------

SYNOPSIS	<pre> STATUS usbSpeakerDynamicAttachRegister (USB_SPKR_ATTACH_CALLBACK callback, /* new callback to be registered */ pVOID arg /* user-defined arg to callback */) </pre>
DESCRIPTION	<p><i>callback</i> is a caller-supplied function of the form:</p> <pre> typedef (*USB_SPKR_ATTACH_CALLBACK) (pVOID arg, SEQ_DEV *pSeqDev, UINT16 attachCode); </pre> <p>usbSpeakerLib will invoke <i>callback</i> each time a USB audio device is attached to or removed from the system. <i>arg</i> is a caller-defined parameter which will be passed to the <i>callback</i> each time it is invoked. The <i>callback</i> will also be passed a pointer to the SEQ_DEV structure for the channel being created/destroyed and an attach code of USB_SPKR_ATTACH or USB_SPKR_REMOVE.</p>
RETURNS	OK, or ERROR if unable to register callback
ERRNO	<p>S_usbSpeakerLib_BAD_PARAM Bad Parameter is recieved</p> <p>S_usbSpeakerLib_OUT_OF_MEMORY Sufficient memory is not available</p>
SEE ALSO	usbSpeakerLib

usbSpeakerDynamicAttachUnregister()

NAME	usbSpeakerDynamicAttachUnregister() – Unregisters speaker attach callback
SYNOPSIS	<pre> STATUS usbSpeakerDynamicAttachUnRegister (USB_SPKR_ATTACH_CALLBACK callback, /* callback to be unregistered */ pVOID arg /* user-defined arg to callback */) </pre>
DESCRIPTION	<p>This function cancels a previous request to be dynamically notified for audio device attachment and removal. The <i>callback</i> and <i>arg</i> paramters must exactly match those passed in a previous call to usbSpeakerDynamicAttachRegister().</p>
RETURNS	OK, or ERROR if unable to unregister callback

ERRNO **S_usbSpeakerLib_NOT_REGISTERED**
 Could not register the attachment callback function

SEE ALSO **usbSpeakerLib**

usbSpeakerSeqDevLock()

NAME **usbSpeakerSeqDevLock()** – Marks SEQ_DEV structure as in use

SYNOPSIS

```
STATUS usbSpeakerSeqDevLock
(
    SEQ_DEV *pChan /* SEQ_DEV to be marked as in use */
)
```

DESCRIPTION A caller uses **usbSpeakerSeqDevLock()** to notify **usbSpeakerLib** that it is using the indicated SEQ_DEV structure. **usbSpeakerLib** maintains a count of callers using a particular SEQ_DEV structure so that it knows when it is safe to dispose of a structure when the underlying USB speaker is removed from the system. So long as the "lock count" is greater than zero, **usbSpeakerLib** will not dispose of an SEQ_DEV structure.

RETURNS OK, or ERROR if unable to mark SEQ_DEV structure in use.

ERRNO none

SEE ALSO **usbSpeakerLib**

usbSpeakerSeqDevUnlock()

NAME **usbSpeakerSeqDevUnlock()** – Marks SEQ_DEV structure as unused

SYNOPSIS

```
STATUS usbSpeakerSeqDevUnlock
(
    SEQ_DEV *pChan /* SEQ_DEV to be marked as unused */
)
```

DESCRIPTION This function releases a lock placed on an SEQ_DEV structure. When a caller no longer needs an SEQ_DEV structure for which it has previously called **usbSpeakerSeqDevLock()**, then it should call this function to release the lock.

NOTE	If the underlying USB speaker device has already been removed from the system, then this function will automatically dispose of the SEQ_DEV structure if this call removes the last lock on the structure. Therefore, a caller must not reference the SEQ_DEV again structure after making this call.
RETURNS	OK, or ERROR if unable to mark SEQ_DEV structure unused
ERRNO	S_usbSpeakerLib_NOT_LOCKED No lock to unlock
SEE ALSO	usbSpeakerLib

usbTargControlPayloadRcv()

NAME	usbTargControlPayloadRcv() – receives data on the default control pipe
SYNOPSIS	<pre> STATUS usbTargControlPayloadRcv (USB_TARG_CHANNEL targChannel, /* target channel */ UINT16 bfrLen, /* length of data to be received */ pUINT8 pBfr, /* ptr to bfr */ ERP_CALLBACK userCallback /* USB Target Applcaition Callback */) </pre>
DESCRIPTION	USB Targlib Layer automatically creates a pipe to manage communication on the default control pipe (#0) defined by the USB. Certain application callbacks may need to receive additional data on the control OUT endpoint in order to complete processing of the control pipe request. This function allows a caller to receive data on a control pipe.
RETURNS	OK, or ERROR if unable to submit ERP to receive additional data
ERRNO	S_usbTargLib_GENERAL_FAULT Fault occurred in upper layers. S_usbTargLib_BAD_PARAM Bad Parameter is passed.
SEE ALSO	usbTargDefaultPipe

usbTargControlResponseSend()

NAME	usbTargControlResponseSend() – sends data to host on the control pipe
SYNOPSIS	<pre>STATUS usbTargControlResponseSend (USB_TARG_CHANNEL targChannel, /* target channel */ UINT16 bfrLen, /* length of response 0 */ pUINT8 pBfr /* ptr to bfr */)</pre>
DESCRIPTION	The USB Target Layer automatically creates a pipe to manage communication on the default control endpoint (#0) defined by the USB. Certain application callbacks may need to formulate a response and send it to the host. This function allows a caller to respond to a host control pipe request.* This function returns as soon as the transfer is enqueued.
RETURNS	OK, or ERROR if unable to submit response to host.
ERRNO	S_usbTargLib_GENERAL_FAULT Fault occurred in upper layers. S_usbTargLib_BAD_PARAM Bad Parameter is passed.
SEE ALSO	usbTargDefaultPipe

usbTargControlStatusSend()

NAME	usbTargControlStatusSend() – sends control transfer status to the host
SYNOPSIS	<pre>STATUS usbTargControlStatusSend (USB_TARG_CHANNEL targChannel /* target channel */)</pre>
DESCRIPTION	This function is used to send the status to the host. This function is used when the control transfer does not have a data stage.
RETURNS	OK, or ERROR if unable to submit the status ERP.
ERRNO	S_usbTargLib_GENERAL_FAULT Fault occurred in upper layers.

S_usbTargLib_BAD_PARAM
 Bad Parameter is passed.

SEE ALSO **usbTargDefaultPipe**

usbTargCurrentFrameGet()

NAME **usbTargCurrentFrameGet()** – retrieves the current USB frame number

SYNOPSIS

```
STATUS usbTargCurrentFrameGet
(
    USB_TARG_CHANNEL targChannel, /* target channel */
    pUINT16           pFrameNo    /* current frame number */
)
```

DESCRIPTION This function allows a caller to retrieve the current USB frame number for the bus to which *targChannel* is connected. Upon return, the current frame number is stored in *pFrameNo*.

RETURNS OK, or **ERROR** if unable to retrieve USB frame number

ERRNO **S_usbTargLib_TCD_FAULT**
 Fault occurred in TCD

SEE ALSO **usbTargDeviceControl**

usbTargDeviceFeatureClear()

NAME **usbTargDeviceFeatureClear()** – clears a specific feature

SYNOPSIS

```
STATUS usbTargDeviceFeatureClear
(
    USB_TARG_CHANNEL targChannel, /* target channel */
    UINT16           ufeatureSelector /* feature to be cleared */
)
```

DESCRIPTION This function is used to clear a device specific feature.

RETURNS OK or **ERROR** if not able to clear the feature.

ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD.
SEE ALSO	usbTargDeviceControl

usbTargDeviceFeatureSet()

NAME	usbTargDeviceFeatureSet() – sets or enable a specific feature
SYNOPSIS	<pre>STATUS usbTargDeviceFeatureSet (USB_TARG_CHANNEL targChannel, /* target channel */ UINT16 ufeatureSelector, /* feature to be set */ UINT8 uTestSelector /* test selector value */)</pre>
DESCRIPTION	This function is used to set or enable a device specific feature.
RETURNS	OK or ERROR if not able to set the feature.
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD
SEE ALSO	usbTargDeviceControl

usbTargDisable()

NAME	usbTargDisable() – disables a target channel
SYNOPSIS	<pre>STATUS usbTargDisable (USB_TARG_CHANNEL targChannel /* target to disable */)</pre>
DESCRIPTION	This function is the counterpart to the usbTargEnable() function. This function disables the indicated target channel.
RETURNS	OK, or ERROR if unable to disable the target channel.

ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD.
	S_usbTargLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTargInitExit

usbTargEnable()

NAME	usbTargEnable() – enables target channel onto USB
SYNOPSIS	<pre>STATUS usbTargEnable (USB_TARG_CHANNEL targChannel /* target to enable */)</pre>
DESCRIPTION	After attaching a TCD to usbTargLib and performing any other application- specific initialization that might be necessary, this function should be called to enable a target channel. The USB target controlled by the TCD will not appear as a device on the USB until this function has been called.
RETURNS	OK, or ERROR if unable to enable target channel.
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD.
	S_usbTargLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTargInitExit

usbTargInitialize()

NAME	usbTargInitialize() – initializes the USB Target Library
SYNOPSIS	<pre>STATUS usbTargInitialize (void)</pre>
DESCRIPTION	This routine is used to initialize the USB Target Library. It initializes the OS library, creates the handles and mutexes.

RETURNS	OK or ERROR
ERRNO	S_usbTargLib_GENERAL_FAULT Fault occured in software layers. S_usbTargLib_OUT_OF_RESOURCES Sufficient resources are not available.
SEE ALSO	usbTargInitExit

usbTargKbdCallbackInfo()

NAME	usbTargKbdCallbackInfo() – returns usbTargKbdLib callback table
SYNOPSIS	<pre>VOID usbTargKbdCallbackInfo (struct usbTargCallbackTable ** ppCallbacks, /* Callback table pointer */ pVOID *pCallbackParam /* target app-specific parameter */)</pre>
DESCRIPTION	This function is called by the initialization routine. It returns the callback table information.
RETURNS	N/A
ERRNO	none.
SEE ALSO	usbTargKbdLib

usbTargKbdInjectReport()

NAME	usbTargKbdInjectReport() – injects a "boot report"
SYNOPSIS	<pre>STATUS usbTargKbdInjectReport (PHID_KBD_BOOT_REPORT pReport, /* Boot Report to be injected */ UINT16 reportLen /* Length of the boot report */)</pre>

DESCRIPTION	This function injects the boot report into the interrupt pipe. <i>pReport</i> is the pointer to the boot report to be injected. <i>reportErpCallback</i> is called after the boot report is successfully sent to the host.
RETURNS	OK, or ERROR if unable to inject report
ERRNO	none.
SEE ALSO	usbTargKbdLib

usbTargMgmtCallback()

NAME	usbTargMgmtCallback() – invoked when HAL detects a management event
SYNOPSIS	<pre>STATUS usbTargMgmtCallback (pVOID pTargTcd, /* pointer to TARG_TCD structure */ UINT16 mngmtCode, /* management event code */ pVOID pContext /* parameter of management event */)</pre>
DESCRIPTION	This function is invoked by the HAL when the HAL detects a "management" event on a target channel.
RETURNS	OK or ERROR if there is an error in handling the management event.
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD
SEE ALSO	usbTargDeviceControl

usbTargMsCallbackInfo()

NAME	usbTargMsCallbackInfo() – returns usbTargPrnLib callback table
SYNOPSIS	<pre>VOID usbTargMsCallbackInfo (struct usbTargCallbackTable ** ppCallbacks, /*USB_TARG_CALLBACK_TABLE */</pre>

```
pVOID * pCallbackParam /* Callback Parameter */
)
```

DESCRIPTION	This function returns the callback table pointer .
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargMsLib

usbTargPipeCreate()

NAME **usbTargPipeCreate()** – creates a pipe for communication on an endpoint

SYNOPSIS

```
STATUS usbTargPipeCreate
(
    USB_TARG_CHANNEL    targChannel,          /* target channel */
    pUSB_ENDPOINT_DESCR pEndpointDesc,        /* USB_ENDPOINT_DESCR */
    UINT16               uConfigurationValue,  /* configuration value */
    UINT16               uInterface,           /* Number of interface which */
                                           /* holds this endpoint */
    UINT16               uAltSetting,          /* alternate Setting */
    pUSB_TARG_PIPE       pPipeHandle           /* pointer to pipe handle */
)
```

DESCRIPTION	This function creates a pipe for communication on an endpoint attached to a specific target endpoint. In return we get the pipe handle which is used by that endpoint for communication.
RETURNS	OK, or ERROR if unable to create pipe
ERRNO	S_usbTargLib_TCD_FAULT Fault occured in TCD. S_usbTargLib_BAD_PARAM Bad parameter is passed. S_usbTargLib_OUT_OF_RESOURCES Sufficient resources not available.
SEE ALSO	usbTargPipeFunc

usbTargPipeDestroy()

NAME	usbTargPipeDestroy() – destroys an endpoint pipe
SYNOPSIS	<pre>STATUS usbTargPipeDestroy (USB_TARG_PIPE pipeHandle /* pipe to be destroyed */)</pre>
DESCRIPTION	This function tears down a pipe previously created by calling usbTargPipeCreate() .
RETURNS	OK, or ERROR if unable to destroy pipe.
ERRNO	S_usbTargLib_TCD_FAULT Error occurred in TCD.
SEE ALSO	usbTargPipeFunc

usbTargPipeStatusGet()

NAME	usbTargPipeStatusGet() – returns the endpoint status
SYNOPSIS	<pre>STATUS usbTargPipeStatusGet (USB_TARG_PIPE pipeHandle, /* Handle to the pipe */ pUINT8 pBuf /* Buffer to hold the pipe status */)</pre>
DESCRIPTION	This function is used to get the status of the pipeas per GET_STATUS request. The status of the pipe is stored in the pointer variable pBuf
RETURNS	OK, or ERROR if unable to get state
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD.
SEE ALSO	usbTargPipeFunc

usbTargPipeStatusSet()

NAME	usbTargPipeStatusSet() – sets pipe stalled/unstalled status
SYNOPSIS	<pre>STATUS usbTargPipeStatusSet (USB_TARG_PIPE pipeHandle, /* Handle to the pipe */ UINT16 state /* State of the pipe to be set */)</pre>
DESCRIPTION	<p>If the target application detects an error while servicing a pipe, it may choose to stall the endpoint(s) associated with that pipe.</p> <p>This function allows the caller to set the state of a pipe as "stalled" or "un-stalled".</p>
RETURNS	OK, or ERROR if unable to set indicated state
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD.
SEE ALSO	usbTargPipeFunc

usbTargPrnCallbackInfo()

NAME	usbTargPrnCallbackInfo() – returns usbTargPrnLib callback table
SYNOPSIS	<pre>VOID usbTargPrnCallbackInfo (pUSB_TARG_CALLBACK_TABLE *ppCallbacks, /* Pointer to callback */ /* table */ pVOID *pCallbackParam /* target app-specific */ /* parameter */)</pre>
DESCRIPTION	<p>This function is called by the initialization routine. It returns the information about the callback table.</p>
RETURNS	N/A
ERRNO	none
SEE ALSO	usbTargPrnLib

usbTargPrnDataInfo()

NAME	usbTargPrnDataInfo() – returns buffer status/info
SYNOPSIS	<pre>STATUS usbTargPrnDataInfo (pUINT8 * ppBfr, /* Pointer to the buffer address */ pUINT16 pActLen /* Actual length of the data */)</pre>
DESCRIPTION	This function returns the status the bulk buffer which consist of the data sent by the printer. <i>pActLen</i> will consist of the actual length of data to be printed.
RETURNS	OK if buffer has valid data, else ERROR
ERRNO	none.
SEE ALSO	usbTargPrnLib

usbTargPrnDataRestart()

NAME	usbTargPrnDataRestart() – restarts listening ERP
SYNOPSIS	<pre>STATUS usbTargPrnDataRestart (void)</pre>
DESCRIPTION	This function restarts the listening of ERP on Bulk Out Pipe.
RETURNS	OK, or ERROR if unable to re-initiate ERP
ERRNO	none
SEE ALSO	usbTargPrnLib

usbTargRbcBlockDevCreate()

NAME	usbTargRbcBlockDevCreate() – create an RBC BLK_DEV device.
-------------	-------------------------------------------------------------------

SYNOPSIS	<code>STATUS usbTargRbcBlockDevCreate (void)</code>
DESCRIPTION	This routine creates an RBC BLK I/O device. The RAM driver will be used for the actual implementation.
RETURNS	OK or ERROR , if not able to create the RAM Disk.
ERRNO	none.
SEE ALSO	usbTargRbcCmd

usbTargRbcBlockDevGet()

NAME	usbTargRbcBlockDevGet() – return opaque pointer to the RBC BLK I/O DEV device
SYNOPSIS	<code>pVOID usbTargRbcBlockDevGet (void)</code>
DESCRIPTION	structure. This routine returns an opaque pointer to the RBC BLK I/O DEV device structure.
RETURNS	Pointer to the RBC BLK I/O DEV structure
ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargRbcBlockDevSet()

NAME	usbTargRbcBlockDevSet() – set the pointer to the RBC BLK I/O DEV device structure.
SYNOPSIS	<pre>STATUS usbTargRbcBlockDevSet (pVOID *blkDev /* pointer to the BLK_DEV device */)</pre>
DESCRIPTION	This routine sets the RBC BLK_DEV pointer that is accessed by the usbTargRbcBlockDevGet() routine.

RETURNS OK or ERROR

ERRNO none

SEE ALSO **usbTargRbcCmd**

usbTargRbcBufferWrite()

NAME **usbTargRbcBufferWrite()** – write micro-code to the RBC device

SYNOPSIS

```
STATUS usbTargRbcBufferWrite
(
    UINT8      arg[10], /* the RBC command */
    UINT8 ** ppData, /* micro-code location on device */
    UINT32 * pSize /* size of micro-code location on device */
)
```

DESCRIPTION This routine writes micro-code to the RBC block I/O device.

RETURNS OK or ERROR

ERRNO none.

SEE ALSO **usbTargRbcCmd**

usbTargRbcCacheSync()

NAME **usbTargRbcCacheSync()** – synchronize the cache of the RBC device

SYNOPSIS

```
STATUS usbTargRbcCacheSync
(
    UINT8 arg[10] /* the RBC command */
)
```

DESCRIPTION This routine synchronizes the cache of the RBC block I/O device.

RETURNS OK or ERROR

ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargRbcCapacityRead()

NAME	usbTargRbcCapacityRead() – read the capacity of the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcCapacityRead (UINT8 arg[10], /* RBC command */ UINT8 **ppData, /* point to capacity data */ UINT32 *pSize /* size of capacity */)</pre>
DESCRIPTION	This routine reads the capacity of the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none.
SEE ALSO	usbTargRbcCmd

usbTargRbcFormat()

NAME	usbTargRbcFormat() – format the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcFormat (UINT8 arg[6] /* the RBC command */)</pre>
DESCRIPTION	This routine formats the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none

SEE ALSO **usbTargRbcCmd**

usbTargRbcInquiry()

NAME **usbTargRbcInquiry()** – retrieve inquiry data from the RBC device

SYNOPSIS

```
STATUS usbTargRbcInquiry
(
    UINT8  cmd[6],      /* the RBC command */
    UINT8  **ppData,    /* location of inquiry data on device */
    UINT32 *pSize       /* size of inquiry data on device */
)
```

DESCRIPTION This routine retrieves inquiry data from the RBC block I/O device.

RETURNS **OK or ERROR**

ERRNO none

SEE ALSO **usbTargRbcCmd**

usbTargRbcModeSelect()

NAME **usbTargRbcModeSelect()** – select the mode parameter page of the RBC device

SYNOPSIS

```
STATUS usbTargRbcModeSelect
(
    UINT8  arg[6],      /* the RBC command */
    UINT8  ** ppData,   /* location of mode parameter data on device */
    UINT32 * pSize      /* size of mode parameter data on device */
)
```

DESCRIPTION This routine selects the mode parameter page of the RBC block I/O device. For non-removable medium devices the SAVE PAGES (SP) bit shall be set to one. This indicates that the device shall perform the specified MODE SELECT operation and shall save, to a non-volatile vendor-specific location, all the changeable pages, including any sent with the command. Application clients should issue MODE SENSE(6) prior to each MODE SELECT(6) to determine supported pages, page lengths, and other parameters.

RETURNS **OK or ERROR**

ERRNO none.

SEE ALSO [usbTargRbcCmd](#)

usbTargRbcModeSelect10()

NAME [usbTargRbcModeSelect10\(\)](#) – select the mode parameter page of the RBC device

SYNOPSIS

```
STATUS usbTargRbcModeSelect10
(
    UINT8      arg[10], /* the RBC command */
    UINT8 ** ppData, /* location of mode parameter data on device */
    UINT32 * pSize /* size of mode parameter data on device */
)
```

DESCRIPTION This routine selects the mode parameter page of the RBC block I/O device. For non-removable medium devices the SAVE PAGES (SP) bit shall be set to one. This indicates that the device shall perform the specified MODE SELECT operation and shall save, to a non-volatile vendor-specific location, all the changeable pages, including any sent with the command. Application clients should issue MODE SENSE(10) prior to each MODE SELECT(10) to determine supported pages, page lengths, and other parameters.

RETURNS OK or ERROR

ERRNO none.

SEE ALSO [usbTargRbcCmd](#)

usbTargRbcModeSense()

NAME [usbTargRbcModeSense\(\)](#) – retrieve sense data from the RBC device

SYNOPSIS

```
STATUS usbTargRbcModeSense
(
    UINT8      arg[6], /* the RBC command */
    UINT8 ** ppData, /* location mode parameter data on device */
    UINT32 * pSize /* size of mode parameter data on device */
)
```


DESCRIPTION	This routine retrieves sense data from the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none.
SEE ALSO	usbTargRbcCmd

usbTargRbcModeSense10()

NAME	usbTargRbcModeSense10() – request for mode sense 10 command
SYNOPSIS	<pre>STATUS usbTargRbcModeSense10 (UINT8 arg[10], /* the RBC command */ UINT8 ** ppData, /* location mode parameter data on device */ UINT32 * pSize /* size of mode parameter data on device */)</pre>
DESCRIPTION	This routine retrieves sense data from the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none.
SEE ALSO	usbTargRbcCmd

usbTargRbcPersistentReserveIn()

NAME	usbTargRbcPersistentReserveIn() – send reserve data to the host
SYNOPSIS	<pre>STATUS usbTargRbcPersistentReserveIn (UINT8 arg[10], /* the RBC command */ UINT8 ** ppData, /* location of reserve data on device */ UINT32 * pSize /* size of reserve data */)</pre>
DESCRIPTION	This routine requests reserve data to be sent to the initiator.

RETURNS	OK or ERROR
ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargRbcPersistentReserveOut()

NAME	usbTargRbcPersistentReserveOut() – reserve resources on the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcPersistentReserveOut (UINT8 arg[10], /* the RBC command */ UINT8 ** ppData, /* location of reserve data on device */ UINT32 *pSize /* size of reserve data */)</pre>
DESCRIPTION	This routine reserves resources on the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargRbcPreventAllowRemoval()

NAME	usbTargRbcPreventAllowRemoval() – prevent or allow the removal of the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcPreventAllowRemoval (UINT8 arg[6] /* the RBC command */)</pre>
DESCRIPTION	This routine prevents or allows the removal of the RBC block I/O device.
RETURNS	OK or ERROR

ERRNO none

SEE ALSO **usbTargRbcCmd**

usbTargRbcRead()

NAME **usbTargRbcRead()** – read data from the RBC device

SYNOPSIS

```
STATUS usbTargRbcRead
(
    UINT8      arg[10], /* the RBC command */
    UINT8      ** ppData, /* pointer to where data will be read by host */
    UINT32      * pSize /* size of data to be read */
)
```

DESCRIPTION This routine reads data from the RBC block I/O device.

RETURNS **OK** or **ERROR**

ERRNO none
none

SEE ALSO **usbTargRbcCmd**

usbTargRbcRelease()

NAME **usbTargRbcRelease()** – release a resource on the RBC device

SYNOPSIS

```
STATUS usbTargRbcRelease
(
    UINT8 arg[6] /* the RBC command */
)
```

DESCRIPTION This routine releases a resource on the RBC block I/O device.

RETURNS **OK** or **ERROR**

ERRNO none.

SEE ALSO **usbTargRbcCmd**

usbTargRbcRequestSense()

NAME **usbTargRbcRequestSense()** – request sense data from the RBC device

SYNOPSIS

```
STATUS usbTargRbcRequestSense
(
    UINT8      arg[6], /* the RBC command */
    UINT8 ** ppData, /* location of sense data on device */
    UINT32 *pSize /* size of sense data */
)
```

DESCRIPTION This routine requests sense data from the RBC block I/O device.

RETURNS OK or ERROR

ERRNO N/A

SEE ALSO **usbTargRbcCmd**

usbTargRbcReserve()

NAME **usbTargRbcReserve()** – reserve a resource on the RBC device

SYNOPSIS

```
STATUS usbTargRbcReserve
(
    UINT8 arg[6] /* the RBC command */
)
```

DESCRIPTION This routine reserves a resource on the RBC block I/O device.

RETURNS OK or ERROR

ERRNO none

SEE ALSO **usbTargRbcCmd**

usbTargRbcStartStop()

NAME **usbTargRbcStartStop()** – start or stop the RBC device

SYNOPSIS `STATUS usbTargRbcStartStop`
 (
 UINT8 arg[6] /* the RBC command */
)

DESCRIPTION This routine starts or stops the RBC block I/O device.

RETURNS OK or ERROR

ERRNO none

SEE ALSO **usbTargRbcCmd**

usbTargRbcTestUnitReady()

NAME **usbTargRbcTestUnitReady()** – test if the RBC device is ready

SYNOPSIS `STATUS usbTargRbcTestUnitReady`
 (
 UINT8 arg[6] /* the RBC command */
)

DESCRIPTION This routine tests whether the RBC block I/O device is ready.

RETURNS OK or ERROR

ERRNO none.

SEE ALSO **usbTargRbcCmd**

usbTargRbcVendorSpecific()

NAME	usbTargRbcVendorSpecific() – vendor specific call
SYNOPSIS	<pre>STATUS usbTargRbcVendorSpecific (UINT8 arg[10], /* the RBC command */ UINT8 ** ppData, /* location of sense data on device */ UINT32 * pSize /* size of sense data */)</pre>
DESCRIPTION	This routine is a vendor specific call.
RETURNS	OK
ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargRbcVerify()

NAME	usbTargRbcVerify() – verify the last data written to the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcVerify (UINT8 arg[10] /* the RBC command */)</pre>
DESCRIPTION	This routine verifies the last data written to the RBC block I/O device.
RETURNS	OK or ERROR.
ERRNO	none.
SEE ALSO	usbTargRbcCmd

usbTargRbcWrite()

NAME	usbTargRbcWrite() – write to the RBC device
SYNOPSIS	<pre>STATUS usbTargRbcWrite (UINT8 arg[10], /* the RBC command */ UINT8 ** ppData, /* location where data will be written to device */ UINT32 *pSize /* size of location on device */)</pre>
DESCRIPTION	This routine writes to the RBC block I/O device.
RETURNS	OK or ERROR
ERRNO	none
SEE ALSO	usbTargRbcCmd

usbTargSetupErpCallback()

NAME	usbTargSetupErpCallback() – handles the setup packet
SYNOPSIS	<pre>VOID usbTargSetupErpCallback (pUSB_ERP pErp /* Pointer to ERP structure */)</pre>
DESCRIPTION	This function is called when a setup packet is received.
RETURNS	N/A
ERRNO	None
SEE ALSO	usbTargDefaultPipe

usbTargShutdown()

NAME	usbTargShutdown() – shutdown the USB target library
SYNOPSIS	STATUS usbTargShutdown (void)
DESCRIPTION	This function is used to shutdown the USB Target Library. It frees the various resources allotted.
RETURNS	OK or ERROR
ERRNO	S_usbTargLib_NOT_INITIALIZED Initialized variable is used. S_usbTargLib_TCD_FAULT Fault occurred in TCD. S_usbTargLib_APP_FAULT Application Specific fault occurred.
SEE ALSO	usbTargInitExit

usbTargSignalResume()

NAME	usbTargSignalResume() – drives RESUME signalling on USB
SYNOPSIS	STATUS usbTargSignalResume (USB_TARG_CHANNEL targChannel /* target channel */)
DESCRIPTION	If a USB is in the SUSPENDED state, it is possible for a device (target) to request the bus to wake up (called remote wakeup). This function allows the caller to drive USB resume signalling. The function will return after resume signalling has completed.
RETURNS	OK, or ERROR if unable to drive RESUME signalling
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD
SEE ALSO	usbTargDeviceControl

usbTargTcdAttach()

NAME	usbTargTcdAttach() – to attach the TCD to the target library
SYNOPSIS	<pre> STATUS usbTargTcdAttach (USB_TCD_EXEC_FUNC tcdExecFunc, /* single entry point of the TCD */ pVOID tcdParam, /* parameter passed to TCD */ pUSB_TARG_CALLBACK_TABLE pCallbacks, /* pointer to Callback functions */ pVOID callbackParam, /* parameter to callback functions */ pUSB_TARG_CHANNEL pTargChannel /* target channel handle returned */) </pre>
DESCRIPTION	This function is used to attach the TCD to the Target Library. In response to a successful TCD attachment, usbTargLib returns a USB_TARG_CHANNEL handle to the caller. This handle must be used in all subsequent calls to usbTargLib to identify a given target channel.
RETURNS	OK or ERROR
ERRNO	<p>S_usbTargLib_OUT_OF_MEMORY Memory not present to allocate variables.</p> <p>S_usbTargLib_TCD_FAULT Fault occurred in TCD.</p> <p>S_usbTargLib_OUT_OF_RESOURCES Sufficient resources not available.</p> <p>S_usbTargLib_BAD_PARAM Bad parameter is passed.</p> <p>S_usbTargLib_APP_FAULT Application Specific Fault occurred.</p>
SEE ALSO	usbTargInitExit

usbTargTcdDetach()

NAME	usbTargTcdDetach() – detaches a USB target controller driver
SYNOPSIS	STATUS usbTargTcdDetach

```
(
    USB_TARG_CHANNEL targChannel /* handle to target channel */
)
```

DESCRIPTION	This function detaches a USB TCD which was previously attached to the usb Target Library by calling usbTargTcdAttach() . <i>targChannel</i> is the handle of the target channel originally returned by usbTargTcdAttach() .
RETURNS	OK, or ERROR if unable to detach TCD.
ERRNO	S_usbTargLib_TCD_FAULT Fault occured in TCD. S_usbTargLib_BAD_PARAM Bad parameter is passed. S_usbTargLib_APP_FAULT Application Specific Fault occured.
SEE ALSO	usbTargInitExit

usbTargTransfer()

NAME	usbTargTransfer() – to transfer data through a pipe
SYNOPSIS	<pre>STATUS usbTargTransfer (USB_TARG_PIPE pipeHandle, /* handle to the pipe */ pUSB_ERP pErp /* ERP to be transfered */)</pre>
DESCRIPTION	This function is used to initiate an transfer on the pipe indicated by <i>pipeHandle</i> . The transfer is described by an ERP, or endpoint request packet, which must be allocated and initialized by the caller prior to invoking usbTargTransfer() .
RETURNS	OK or Error if not able to transfer data.
ERRNO	S_usbTargLib_TCD_FAULT Fault occured in TCD. S_usbTargLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTargPipeFunc

usbTargTransferAbort()

NAME	usbTargTransferAbort() – cancels a previously submitted USB_ERP
SYNOPSIS	<pre>STATUS usbTargTransferAbort (USB_TARG_PIPE pipeHandle, /* pipe for transfer to abort */ pUSB_ERP pErp /* ERP to be aborted */)</pre>
DESCRIPTION	This function aborts an ERP which was previously submitted through a call to usbTargTransfer() .
RETURNS	OK, or ERROR if unable to cancel USB_ERP
ERRNO	S_usbTargLib_TCD_FAULT Fault occurred in TCD. S_usbTargLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTargPipeFunc

usbTcdIsp1582EvalExec()

NAME	usbTcdIsp1582EvalExec() – single Entry Point for ISP 1582 TCD
SYNOPSIS	<pre>STATUS usbTcdIsp1582EvalExec (pVOID pTrb /* TRB to be executed */)</pre>
DESCRIPTION	This is the single entry point for the Philips ISP 1582 USB TCD (Target Controller Driver). The function qualifies the TRB passed by the caller and fans out to the appropriate TCD function handler.
RETURNS	OK or ERROR if failed to execute TRB passed by caller.
ERRNO	S_usbTcdLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTcdIsp1582InitExit

usbTcdNET2280Exec()

NAME	usbTcdNET2280Exec() – single Entry Point for NETCHIP 2280 TCD
SYNOPSIS	<pre>STATUS usbTcdNET2280Exec (pVOID pTrb /* TRB to be executed */)</pre>
DESCRIPTION	This is the single entry point for the NETCHIP 2280 USB TCD (Target Controller Driver). The function qualifies the TRB passed by the caller and fans out to the appropriate TCD function handler.
RETURNS	OK or ERROR if failed to execute TRB passed by caller.
ERRNO	S_usbTcdLib_BAD_PARAM Bad parameter is passed.
SEE ALSO	usbTcdNET2280InitExit

usbTcdPdiusbd12EvalExec()

NAME	usbTcdPdiusbd12EvalExec() – single entry point for PDIUSB12 TCD
SYNOPSIS	<pre>STATUS usbTcdPdiusbd12EvalExec (pVOID pTrb /* TRB to be executed */)</pre>
DESCRIPTION	This is the single entry point for the Philips PDIUSB12 (ISA eval version) USB TCD (Target Controller Driver). The function qualifies the TRB passed by the caller and fans out to the appropriate TCD function handler.
RETURNS	OK or ERROR if failed to execute TRB passed by caller.
ERRNO	none.
SEE ALSO	usbTcdPdiusbd12InitExit

usbTransferTime()

NAME **usbTransferTime()** – Calculates the bus time required for a USB transfer.

SYNOPSIS

```
UINT32 usbTransferTime
(
    UINT16 transferType,    /* transfer type */
    UINT16 direction,      /* transfer direction */
    UINT16 speed,          /* speed of pipe */
    UINT32 bytes,          /* number of bytes for packet to be calc'd */
    UINT32 hostDelay,      /* host controller delay per packet */
    UINT32 hostHubLsSetup  /* host controller time for low-speed setup */
)
```

DESCRIPTION This function calculates the time a transfer of a given number of bytes will require on the bus, measured in nanoseconds (10E-9 seconds). The formulas used here are taken from Section 5.9.3 of Revision 1.1 of the USB spec.

transferType, *direction*, and *speed* should describe the characteristics of the pipe/transfer as USB_XFRTYPE_xxxx, USB_DIR_xxxx, and USB_SPEED_xxxx, respectively. *bytes* is the size of the packet for which the transfer time should be calculated. *hostDelay* and *hostHubLsSetup* are the host delay and low-speed hub setup times in nanoseconds, respectively, and are host-controller specific.

RETURNS Worst case number of nanoseconds required for transfer

ERRNO None

SEE ALSO **usbLib**

usbUhcdExit()

NAME **usbUhcdExit()** – uninitialize the USB UHCI Host Controller Driver.

SYNOPSIS USBHST_STATUS usbUhcdExit (void)

DESCRIPTION This function uninitialize the USB UHCD Host Controller Driver and detaches it from the usbd interface layer.

RETURNS USBHST_SUCCESS, USBHST_FALIURE if the UHCD Host Controller uninitializatoin fails

ERRNO	None.
SEE ALSO	usbUhcInitialization

usbUhcInit()

NAME	usbUhcInit() – initialise the USB UHCI Host Controller Driver.
SYNOPSIS	<code>USBHST_STATUS usbUhcInit (void)</code>
DESCRIPTION	<p>This function initializes internal data structures in the UHCI Host Controller Driver. This routine is typically called prior the the vxBus invocation of the device connect.</p> <p>This function registers the UHCI HCD with the USB Layer.</p>
RETURNS	<code>USBHST_SUCCESS</code> or <code>USBHST_FALIURE</code> - if the UHCD Host Controller initialization fails
ERRNO	None.
SEE ALSO	usbUhcInitialization

usbUhcInstantiate()

NAME	usbUhcInstantiate() – instantiate the USB UHCI Host Controller Driver.
SYNOPSIS	<code>VOID usbUhcInstantiate (void)</code>
DESCRIPTION	<p>This routine instantiates the UHCI Host Controller Driver and allows the UHCI Controller driver to be included with the vxWorks image and not be registered with vxBus. UHCI devices will remain orphan devices until the usbUhcInit() routine is called. This supports the <code>INCLUDE_UHCI</code> behaviour of previous vxWorks releases.</p> <p>The routine itself does nothing.</p>
RETURNS	N/A

ERRNO None.

SEE ALSO **usbUhcdInitialization**

usbVxbRootHubAdd()

NAME **usbVxbRootHubAdd()** – configures the root hub

SYNOPSIS **VOID** usbVxbRootHubAdd
 (
 VXB_DEVICE_ID pDevInfo /* struct vxDev* */
)

DESCRIPTION This function configures the root hub. The function is called by vxBus with **VXB_DEVICE_ID** as its parameter. The routine will call the routine that is registered with the USB D to configure the root hub device.

RETURNS none

ERRNO none

SEE ALSO **usb d**

usbVxbRootHubRemove()

NAME **usbVxbRootHubRemove()** – removes the root hub

SYNOPSIS **VOID** usbVxbRootHubRemove
 (
 VXB_DEVICE_ID pDevInfo /* struct vxDev* */
)

DESCRIPTION This function is removes the root hub. The function is called by vxBus with **VXB_DEVICE_ID** as its parameter. The routine will call the routine that is registered with the USB D to remove the root hub device.

RETURNS none

ERRNO none

SEE ALSO **usb**

usbAddressGet()

NAME **usbAddressGet()** – Gets the USB address for a given device.

SYNOPSIS

```
STATUS usbAddressGet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID       nodeId,       /* Node Id of device/hub */
    pUINT16            pDeviceAddress /* Currently assigned device address */
)
*/
```

DESCRIPTION This routine returns the USB address assigned to device specified by *nodeId*.

RETURNS **OK**, or **ERROR**

ERRNO none

SEE ALSO **usbTransUnitMisc**

usbAddressSet()

NAME **usbAddressSet()** – Sets the USB address for a given device.

SYNOPSIS

```
STATUS usbAddressSet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID       nodeId,       /* Node Id of device/hub */
    UINT16             deviceAddress /* New device address */
)
*/
```

DESCRIPTION This routine sets the USB address at which a device will respond to future requests. Upon return, the address of the device identified by *nodeId* will be changed to the value specified in *deviceAddress*. *deviceAddress* must be in the range from zero through 127. The *deviceAddress* must also be unique within the scope of each USB host controller.

The USBD manages USB device addresses automatically, and this routine should never be called by normal USBD clients. Changing a device address may cause serious problems, including device address conflicts, and may cause the USB to cease operation.

RETURNS OK, or ERROR

ERRNO none

SEE ALSO usbTransUnitMisc

usbdBusCountGet()

NAME usbdBusCountGet() – Gets the number of USBs attached to the host.

SYNOPSIS

```
STATUS usbdBusCountGet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    pUINT16              pBusCount   /* Word bfr to receive bus count */
)
```

DESCRIPTION This routine returns the total number of USB host controllers in the system. Each host controller has its own root hub as required by the USB specification. Clients planning to enumerate USB devices using the bus enumeration routines need to know the number of host controllers in order to retrieve the node IDs for each root hub.

pBusCount must point to a UINT16 variable in which the total number of USB host controllers will be stored.

NOTE The number of USB host controllers is not constant. Bus controllers can be added by calling **usbdHcdAttach()** and removed by calling **usbdHcdDetach()**. Again, the dynamic attach routines deal with these situations automatically, and are the preferred mechanism by which most clients should be informed of device attachment and removal.

RETURNS OK, or ERROR if unable to retrieve bus count

ERRNO none

SEE ALSO usbTransUnitMisc

usbdBusStateSet()

NAME usbdBusStateSet() – Sets bus state, such as SUSPEND or RESUME.

SYNOPSIS

```
STATUS usbdBusStateSet
```

```
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID        nodeId,      /* node ID */
    UINT16              busState     /* new bus state: USBD_BUS_xxxx */
)
```

DESCRIPTION	<p>This function allows a client to set the state of the bus to which the specified <i>nodeId</i> is attached. The desired <i>busState</i> is specified as USBD_BUS_xxxx.</p> <p>Typically, a client will use this function to set a bus to the SUSPEND or RESUME state. Clients must use this capability with care, as it will affect all devices on a given bus, and hence all clients communicating with those devices.</p>
RETURNS	OK, or ERROR if unable to set specified bus state
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbClientRegister()

NAME	usbClientRegister() – Registers a new client with the USB.
SYNOPSIS	<pre>STATUS usbClientRegister (PCHAR pClientName, /* Client name */ USBD_CLIENT_HANDLE pClientHandle /* Client hdl returned by USB */)</pre>
DESCRIPTION	<p>This routine invokes the USB function to register a new client. <i>pClientName</i> should point to a string of not more than USBD_NAME_LEN characters (excluding the terminating NULL) which can be used to uniquely identify the client. If successful, upon return the <i>pClientHandle</i> will be filled with a newly-assigned USBD_CLIENT_HANDLE.</p>
RETURNS	OK, or ERROR if unable to register a new client.
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbClientUnregister()

NAME	usbClientUnregister() – Unregisters a USB client.
SYNOPSIS	<pre>STATUS usbClientUnregister (USBD_CLIENT_HANDLE clientHandle /* Client handle */)</pre>
DESCRIPTION	<p>A client invokes this function to release a previously-assigned USB_CLIENT_HANDLE. The USB client will release all resources allocated to the client, aborting any outstanding URBs which may exist for the client.</p> <p>Once this function has been called with a given <i>clientHandle</i>, the client must not attempt to reuse the indicated <i>clientHandle</i>.</p>
RETURNS	OK , or ERROR if unable to unregister the client.
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbConfigurationGet()

NAME	usbConfigurationGet() – Gets the USB configuration for a device.
SYNOPSIS	<pre>STATUS usbConfigurationGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ pUINT16 pConfiguration /* bfr to receive config value */)</pre>
DESCRIPTION	<p>This function returns the currently selected configuration for the device or hub indicated by <i>nodeId</i>. The current configuration value is returned in the low byte of <i>pConfiguration</i>. The high byte is currently reserved and will be 0.</p>
RETURNS	OK , or ERROR if unable to get configuration
ERRNO	none
SEE ALSO	usbTransUnitStd

usbdConfigurationSet()

NAME	usbdConfigurationSet() – Sets the USB configuration for a device.
SYNOPSIS	<pre>STATUS usbdConfigurationSet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ UINT16 configuration, /* New configuration to be set */ UINT16 maxPower /* max power this config will draw */)</pre>
DESCRIPTION	<p>This function sets the current configuration for the device identified by <i>nodeId</i>. The client should pass the desired configuration value in the low byte of <i>configuration</i>. The high byte is currently reserved and should be zero.</p> <p>The client must also pass the maximum current which will be used by this configuration in <i>maxPower</i>.</p>
RETURNS	OK, or ERROR if unable to set configuration
ERRNO	none
SEE ALSO	usbTransUnitStd

usbdCurrentFrameGet()

NAME	usbdCurrentFrameGet() – Returns the current frame number for a USB.
SYNOPSIS	<pre>STATUS usbdCurrentFrameGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of node on desired USB */ pUINT32 pFrameNo, /* bfr to receive current frame no. */ pUINT32 pFrameWindow /* bfr to receive frame window */)</pre>
DESCRIPTION	<p>It is sometimes necessary for clients to retrieve the current USB frame number for a specified host controller. This routine allows a client to retrieve the current USB frame number for the host controller to which <i>nodeId</i> is connected. Upon return, the current frame number is stored in <i>pFrameNo</i>.</p> <p>If <i>pFrameWindow</i> is not NULL, the USBD will also return the maximum frame scheduling window for the indicated USB host controller. The frame scheduling window is essentially</p>

the number of unique frame numbers tracked by the USB host controller. Most USB host controllers maintain an internal frame count which is a 10- or 11-bit number, allowing them to track typically 1,024 or 2,048 unique frames. When starting an isochronous transfer, a client may wish to specify that the transfer will begin in a specific USB frame. For the given USB host controller, the starting frame number can be no more than *frameWindow* frames from the current *frameNo*.

NOTE	The USBD is capable of simultaneously managing multiple USB host controllers, each of which operates independently. Therefore, it is important that the client specify the correct <i>nodeId</i> when retrieving the current frame number. Typically, a client will be interested in the current frame number for the host controller to which a specific device is attached.
RETURNS	OK, or ERROR if unable to retrieve current frame number
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbDescriptorGet()

NAME **usbDescriptorGet()** – Retrieves a USB descriptor.

SYNOPSIS

```

STATUS usbDescriptorGet
(
    USBD_CLIENT_HANDLE clientHandle,      /* Client handle */
    USBD_NODE_ID       nodeId,            /* Node Id of device/hub */
    UINT8              requestType,       /* specifies type of request */
    UINT8              descriptorType,     /* Type of descriptor */
    UINT8              descriptorIndex,    /* Index of descriptor */
    UINT16             languageId,        /* Language ID */
    UINT16             bfrLen,            /* Max length of data to be returned */
    /*
    pUINT8              pBfr,              /* Pointer to bfr to receive data */
    pUINT16             pActLen            /* bfr to receive actual length */
    )

```

DESCRIPTION A client uses this function to retrieve a descriptor from the USB device identified by *nodeId*. *requestType* is defined as it was documented for the **usbFeatureClear()** routine. *descriptorType* specifies the type of descriptor to be retrieved and must be one of the following values:

USB_DESCR_DEVICE
 Specifies the device descriptor.

usbDescriptorSet()

USB_DESCR_CONFIG

Specifies the configuration descriptor.

USB_DESCR_STRING

Specifies a string descriptor.

USB_DESCR_INTERFACE

Specifies an interface descriptor.

USB_DESCR_ENDPOINT

Specifies an endpoint descriptor.

descriptorIndex is the index of the desired descriptor.

For string descriptors, the *languageId* should specify the desired language for the string. According to the USB specification, string descriptors are returned in Unicode format and the *languageId* should be the 16-bit language ID (LANGID) defined by Microsoft for Windows as described in "Developing International Software for Windows 95 and Windows NT." Please refer to Section 9.6.5 of Revision 1.1 of the USB specification for more detail. For device and configuration descriptors, *languageId* should be zero.

The caller must provide a buffer to receive the descriptor data. *pBfr* is a pointer to a caller-supplied buffer of length *bfrLen*. If the descriptor is too long to fit in the buffer provided, the descriptor will be truncated. If a non-NULL pointer is passed in *pActLen*, the actual length of the data transferred will be stored in *pActLen* upon return.

RETURNS OK, or ERROR if unable to get the descriptor

ERRNO none

SEE ALSO [usbTransUnitStd](#)

usbDescriptorSet()

NAME `usbdDescriptorSet()` – Sets a USB descriptor.

SYNOPSIS

STATUS usbdDescriptorSet

```

(
    USB_CLIENT_HANDLE clientHandle,    /* Client handle */
    USB_NODE_ID       nodeId,         /* Node Id of device/hub */
    UINT8             requestType,    /* selects request type */
    UINT8             descriptorType, /* Type of descriptor */
    UINT8             descriptorIndex, /* Index of descriptor */
    UINT16            languageId,     /* Language ID */
    UINT16            bfrLen,         /* Max length of data to be returned */
    */

```

```

    pUINT8          pBfr          /* Pointer to bfr to receive data */
)

```

DESCRIPTION	A client uses this routine to set a descriptor on the USB device identified by <i>nodeId</i> . The parameters <i>requestType</i> , <i>descriptorType</i> , <i>descriptorIndex</i> , and <i>languageId</i> are the same as those described for the usbDescriptorGet() routine. <i>pBfr</i> is a pointer to a buffer of length <i>bfrLen</i> which contains the descriptor data to be sent to the device.
RETURNS	OK, or ERROR if unable to set descriptor
ERRNO	none
SEE ALSO	usbTransUnitStd

usbDynamicAttachRegister()

NAME **usbDynamicAttachRegister()** – Registers client for dynamic attach notification.

SYNOPSIS

```

STATUS usbDynamicAttachRegister
(
    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 - if vendor specific driver
                                /* FALSE - if class specific driver
                                /* User-supplied callback */
    USBD_ATTACH_CALLBACK attachCallback
)

```

DESCRIPTION Clients call this function to indicate to the USBD that they wish to be notified whenever a device of the indicated class/sub-class/protocol (in the case of class-specific devices) or the device of the indicated vendorid/ deviceid/BcdInfo (in case of vendor-specific devices) is attached to or removed from the USB. A client may specify that it wants to receive notification for an entire device class or only for specific sub-classes within that class.

For class-specific devices: *deviceClass*, *deviceSubClass*, and *deviceProtocol* must specify a USB class/sub-class/protocol combination according to the USB specification. For the clients' convenience, **usbLib.h** automatically includes **usb.h**, which defines a number of USB device classes as **USB_CLASS_xxxx** and **USB_SUBCLASS_xxxx**. A value of **USB_NOTIFY_ALL** in any of these parameters acts like a wildcard and matches any value reported by the device for the corresponding field.

For vendor-specific devices: *deviceClass*, *deviceSubClass*, and *deviceProtocol* must specify a USB vendorId/productId/bcdInfo combination.

vendorSpecific should be set to **TRUE** if the driver is vendor-specific or **FALSE** if it is USB class-specific.

attachCallback must be a non-NULL pointer to a client-supplied callback routine of the form **USBD_ATTACH_CALLBACK**:

```
typedef VOID (*USBD_ATTACH_CALLBACK)
(
    USBD_NODE_ID nodeId,
    UINT16 attachAction,
    UINT16 configuration,
    UINT16 interface,
    UINT16 deviceClass,
    UINT16 deviceSubClass,
    UINT16 deviceProtocol
);
```

Immediately upon registration the client should expect that it may begin receiving calls to the *attachCallback* routine. Upon registration, translation unit will call the *attachCallback* for each device of the specified class which is already attached to the system. Thereafter, the translation unit will call the *attachCallback* whenever a new device of the specified class is attached to the system or a device is removed.

Each time the *attachCallback* is called, translation unit will pass the node ID of the device in *nodeId* and an attach code in *attachAction* which explains the reason for the callback. Attach codes are defined as:

USBD_DYNA_ATTACH

USB is notifying the client that node ID is a device which is now attached to the system.

USBD_DYNA_REMOVE

USB is notifying the client that node ID has been detached (removed) from the system.

When the *attachAction* is **USBD_DYNA_REMOVE** the *nodeId* refers to a node ID which is no longer valid. The client should interrogate its internal data structures and delete any references to the specified Node ID. If the client had outstanding requests to the specified *nodeId*, such as data transfer requests, then the USB will fail those outstanding requests before calling the *attachCallback* to notify the client that the device has been removed. In general, therefore, transfer requests related to removed devices should already be taken care of before the *attachCallback* is called.

As a convenience to the *attachCallback* routine, the USB also passes the *deviceClass*, *deviceSubClass*, and *deviceProtocol* of the attached or removed *nodeId* each time it calls the *attachCallback*. Please note that if multiple callbacks are registered, it must be for a different handle and *deviceClass* / *deviceSubClass* / *deviceProtocol*.

Note that this routine will call only one callback for each attach event. This is different from some previous releases where a single attach could cause multiple callbacks.

RETURNS OK, or **ERROR** if unable to register for attach/removal notification.

ERRNO N/A

SEE ALSO **usbTransUnitInit**

usbDynamicAttachUnRegister()

NAME **usbDynamicAttachUnRegister()** – Unregisters a client for attach notification.

SYNOPSIS

```
STATUS usbDynamicAttachUnRegister
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    UINT16              deviceClass,  /* USB class code */
    UINT16              deviceSubClass, /* USB sub-class code */
    UINT16              deviceProtocol, /* USB device protocol code */
    USBD_ATTACH_CALLBACK attachCallback /* user-supplied callback routine */
)
```

DESCRIPTION This function cancels a client's earlier request to be notified of the attachment and removal of devices in the specified class. *deviceClass*, *deviceSubClass*, *deviceProtocol*, and *attachCallback* are defined for the **usbDynamicAttachRegister()** routine and must match exactly the parameters passed in an earlier call to **usbDynamicAttachRegister**.

RETURNS OK, or **ERROR** if unable to unregister for attach/removal notification.

ERRNO N/A

SEE ALSO **usbTransUnitInit**

usbExit()

NAME **usbExit()** – exits USB2.0

SYNOPSIS `USBHST_STATUS usbExit(void)`

DESCRIPTION	This routine frees up memory allocated for the USB2.0 layer and should only be called when bringing the USB2.0 stack down. The routine also un-registers the hub bus type with vxBus.
RETURNS	USBHST_SUCCESS, USBHST_FAILURE if bus count is not zero
ERRNO	None
SEE ALSO	usbd

usbdFeatureClear()

NAME **usbdFeatureClear()** – Clears a USB feature.

SYNOPSIS

```
STATUS usbdFeatureClear
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID       nodeId,       /* Node Id of device/hub */
    UINT16             requestType,  /* Selects request type */
    UINT16             feature,       /* Feature selector */
    UINT16             index         /* Interface/endpoint index */
)
```

DESCRIPTION This function allows a client to clear a USB feature. *nodeId* specifies the node ID of the desired device and *requestType* specifies whether the feature is related to the device, to an interface, or to an endpoint as follows:

USB_RT_DEVICE
Device

USB_RT_INTERFACE
Interface

USB_RT_ENDPOINT
Endpoint

requestType also specifies if the request is standard, class-specific, or vendor-specific as follows:

USB_RT_STANDARD
Standard

USB_RT_CLASS
Class-specific

USB_RT_VENDOR
Vendor-specific

For example, `USB_RT_STANDARD | USB_RT_DEVICE` in *requestType* specifies a standard device request.

The client must pass the feature selector of the device in *feature*. If *featureType* specifies an interface or endpoint, then *index* must contain the interface or endpoint index. *index* should be zero when *featureType* is `USB_SELECT_DEVICE`.

RETURNS OK, or **ERROR** if unable to clear feature

ERRNO none

SEE ALSO `usbTransUnitStd`

usbFeatureSet()

NAME `usbFeatureSet()` – Sets a USB feature.

SYNOPSIS

```
STATUS usbFeatureSet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID       nodeId,       /* Node Id of device/hub */
    UINT16             requestType,   /* Selects request type */
    UINT16             feature,       /* Feature selector */
    UINT16             index          /* Interface/endpoint index */
)
```

DESCRIPTION This function allows a client to set a USB feature. *nodeId* specifies the node ID of the desired device and *requestType* specifies the nature of the feature as defined for the `usbFeatureClear()` function.

The client must pass the feature selector of the device in *feature*. If *requestType* specifies an interface or endpoint, then *index* must contain the interface or endpoint index. *index* should be zero when *requestType* includes `USB_SELECT_DEVICE`.

RETURNS OK, or **ERROR** if unable to set feature

ERRNO none

SEE ALSO `usbTransUnitStd`

usbHcdAttach()

NAME	usbHcdAttach() – Attaches an HCD to the USB.
SYNOPSIS	<pre>STATUS usbHcdAttach (HCD_EXEC_FUNC hcdExecFunc, /* Ptr to HCD's primary entry point */ void * hcdPciCfgHdr, /* HCD-specific parameter */ pGENERIC_HANDLE pAttachToken /* Token to identify HCD in future */)</pre>
DESCRIPTION	<p>The <i>hcdExecFunc</i> passed by the caller must point to the primary entry point of an HCD as defined below:</p> <pre>typedef UINT16 (*HCD_EXEC_FUNC) (PHRB_HEADER pHrb);</pre>
RETURNS	OK
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbHcdDetach()

NAME	usbHcdDetach() – Detaches an HCD from the USB.
SYNOPSIS	<pre>STATUS usbHcdDetach (GENERIC_HANDLE attachToken /* AttachToken returned */)</pre>
DESCRIPTION	<p>The <i>attachToken</i> must be the attach token originally returned by usbHcdAttach() when it first attached to the HCD.</p>
RETURNS	OK
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbHubPortCountGet()

NAME	usbHubPortCountGet() – Returns the number of ports connected to a hub.
SYNOPSIS	<pre>STATUS usbHubPortCountGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID hubId, /* Node Id for desired hub */ pUINT16 pPortCount /* bfr to receive port count */)</pre>
DESCRIPTION	<p>usbHubPortCountGet() gives clients a way to retrieve the number of downstream ports provided by the specified hub. Clients can also retrieve this information by retrieving configuration descriptors from the hub using the configuration routines described below.</p> <p><i>hubId</i> must be the node ID for the desired USB hub. An error will be returned if <i>hubId</i> does not refer to a hub. <i>pPortCount</i> must point to a UINT16 variable in which the number of ports on the specified hub will be stored.</p>
RETURNS	OK, or ERROR if unable to get the hub port count
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbInit()

NAME	usbInit() – initializes USB2.0
SYNOPSIS	<pre>USBHST_STATUS usbInit(void)</pre>
DESCRIPTION	<p>This routine initializes the global variables for the USB2.0 layer. It should be called before any hub, hcd, or class driver initialization code.</p>
RETURNS	USBHST_SUCCESS, USBHST_FAILURE if event's could not be created
ERRNO	None
SEE ALSO	usb

usbInitialize()

NAME	usbInitialize() – Initializes the USB.
SYNOPSIS	<code>STATUS usbInitialize (void)</code>
DESCRIPTION	usbInitialize() must be called at least once before calling other USB functions. usbInitialize() prepares the USB and translation unit to process URBs. Calls to usbInitialize() may be nested, allowing multiple USB clients to be written independently.
RETURNS	OK, or ERROR if the initialization failed.
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbInterfaceGet()

NAME	usbInterfaceGet() – Retrieves the current interface of a device.
SYNOPSIS	<pre>STATUS usbInterfaceGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ UINT16 interfaceIndex, /* Index of interface */ pUINT16 pAlternateSetting /* Current alternate setting */)</pre>
DESCRIPTION	This routine allows a client to query the current alternate setting for a given device's interface. <i>nodeId</i> and <i>interfaceIndex</i> specify the device and interface to be queried, respectively. <i>pAlternateSetting</i> points to a UINT16 variable in which the alternate setting will be stored upon return.
RETURNS	OK, or ERROR if unable to get the interface
ERRNO	none
SEE ALSO	usbTransUnitStd

usbInterfaceSet()

NAME	usbInterfaceSet() – Sets the current interface of a device.
SYNOPSIS	<pre> STATUS usbInterfaceSet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ UINT16 interfaceIndex, /* Index of interface */ UINT16 alternateSetting /* Alternate setting */) </pre>
DESCRIPTION	This routine allows a client to select an alternate setting for a given device's interface. <i>nodeId</i> and <i>interfaceIndex</i> specify the device and interface to be modified, respectively. <i>alternateSetting</i> specifies the new alternate setting.
RETURNS	OK, or ERROR if unable to set the interface
ERRNO	none
SEE ALSO	usbTransUnitStd

usbMngmtCallbackSet()

NAME	usbMngmtCallbackSet() – sets a management callback for a client.
SYNOPSIS	<pre> STATUS usbMngmtCallbackSet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_MNGMT_CALLBACK mngmtCallback, /* management callback */ pVOID mngmtCallbackParam /* client-defined parameter */) </pre>
DESCRIPTION	<p>Management callbacks provide a mechanism for the USBD to inform clients of asynchronous management events on the USB. For example, if the USB is in the SUSPEND state - see usbBusStateSet() - and a USB device drives RESUME signalling, that event can be reported to a client through its management callback.</p> <p><i>clientHandle</i> is a client's registered handle with the USBD. <i>mngmtCallback</i> is the management callback routine of type USB_MNGMT_CALLBACK invoked by the USBD when management events are detected. <i>mngmtCallbackParam</i> is a client-defined parameter passed to the <i>mngmtCallback</i> each time it is invoked. Passing a <i>mngmtCallback</i> of NULL cancels management event callbacks.</p>

When the *mgmtCallback* is invoked, the USBD will also pass to it the **USBD_NODE_ID** of the root node on the bus for which the management event has been detected and a code signifying the type of management event as **USBD_MNGMT_xxxx**.

Clients are not required to register a management callback routine. Clients that do use a management callback are permitted to register only one management callback per **USBD_CLIENT_HANDLE**.

RETURNS OK, or **ERROR** if unable to register management callback

ERRNO N/A

SEE ALSO **usbTransUnitInit**

usbNodeIdGet()

NAME **usbNodeIdGet()** – Gets the ID of a node connected to a hub port.

SYNOPSIS

```
STATUS usbNodeIdGet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID        hubId,       /* Node Id for desired hub */
    UINT16              portIndex,   /* Port index */
    pUINT16              pNodeType,   /* bfr to receive node type */
    pUSBD_NODE_ID        pNodeId     /* bfr to receive Node Id */
)
```

DESCRIPTION Clients use this routine to retrieve the node IDs of the devices attached to each port of a hub. *hubId* and *portIndex* identify the hub and port to which a device may be attached. *pNodeType* must point to a **UINT16** variable to receive a type code as follows:

USB_NODETYPE_NONE

No device is attached to the specified port.

USB_NODETYPE_HUB

A hub is attached to the specified port.

USB_NODETYPE_DEVICE

A non-hub device is attached to the specified port.

If the node type is returned as **USBD_NODE_TYPE_NONE**, then a node ID is not returned and the value returned in *pNodeId* is undefined. If the node type indicates a hub or device is attached to the port, then *pNodeId* will contain the node ID of that hub or device upon return.

RETURNS OK, or **ERROR** if unable to get node ID

ERRNO none

SEE ALSO **usbTransUnitMisc**

usbdNodeInfoGet()

NAME **usbdNodeInfoGet()** – Returns information about a USB node.

SYNOPSIS

```
STATUS usbdNodeInfoGet
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID       nodeId,       /* Node Id of device/hub */
    pUSBD_NODE_INFO    pNodeInfo,    /* Structure to receive node info */
    UINT16             infoLen       /* Len of bfr allocated by client */
)
```

DESCRIPTION This routine retrieves information about the USB device specified by *nodeId*. The USBD copies node information into the *pNodeInfo* structure provided by the caller. This structure is of the form **USBD_NODEINFO** as shown below:

```
typedef struct usbd_nodeinfo
{
    UINT16 nodeType;
    UINT16 nodeSpeed;
    USBD_NODE_ID parentHubId;
    UINT16 parentHubPort;
    USBD_NODE_ID rootId;
} USBD_NODEINFO, *pUSBD_NODEINFO;
```

nodeType specifies the type of node identified by *nodeId* and is defined as **USB_NODETYPE_xxxx**. *nodeSpeed* identifies the speed of the device and is defined as **USB_SPEED_xxxx**. This field is not updated. *parentHubId* and *parentHubPort* identify the node ID and port of the hub to which the indicated node is attached upstream. If the indicated *nodeId* happens to be a root hub, then *parentHubId* and *parentHubPort* will both be zero.

Similarly, *rootId* identifies the node ID of the root hub for the USB to which *nodeId* is attached. If *nodeId* itself happens to be the root hub, then the same value will be returned in *rootId*.

This structure may grow. To provide backwards compatibility, the client must pass the total size of the **USBD_NODEINFO** structure it has allocated in *infoLen*. The USBD will copy fields into this structure only up to the *infoLen* indicated by the caller.

RETURNS **OK**, or **ERROR** if unable to retrieve node information

ERRNO None

SEE ALSO

usbTransUnitMisc

usbPipeCreate()

NAME

usbPipeCreate() – Creates a USB pipe for subsequent transfers.

SYNOPSIS

```
STATUS usbPipeCreate
(
    USBD_CLIENT_HANDLE clientHandle,    /* Client handle */
    USBD_NODE_ID       nodeId,          /* Node Id of device/hub */
    UINT16             endpoint,        /* Endpoint address */
    UINT16             configuration,    /* config w/which pipe associated */
    UINT16             interface,       /* interface w/which pipe associated */
    /*
    UINT16             transferType,     /* Type of transfer: control,
bulk... */
    UINT16             direction,        /* Specifies IN or OUT endpoint */
    UINT16             maxPayload,       /* Maximum data payload per packet */
    /*
    UINT32             bandwidth,        /* Bandwidth required for pipe */
    UINT16             serviceInterval, /* Required service interval */
    pUSBD_PIPE_HANDLE pPipeHandle       /* pipe handle returned by USBSD */
)
```

DESCRIPTION

This routine establishes a pipe which can then be used by a client to exchange data with a USB device endpoint.

nodeId and *endpoint* identify the device and device endpoint, respectively, to which the pipe should be connected. *configuration* and *interface* specify the configuration and interface with which the pipe is associated.

transferType specifies the type of data transfers for which this pipe will be used:

USB_XFRTYPE_CONTROL

Control transfer pipe (message)

USB_XFRTYPE_ISOCH

Isochronous transfer pipe (stream)

USB_XFRTYPE_INTERRUPT

Interrupt transfer pipe (stream)

USB_XFRTYPE_BULK

Bulk transfer pipe (stream)

direction specifies the direction of the pipe as:

USB_DIR_IN

Data moves from device to host.

USB_DIR_OUT

Data moves from host to device.

USB_DIR_INOUT

Data moves bidirectionally (message pipes only).

If the *direction* is specified as **USB_DIR_INOUT**, the USB D assumes that both the in and out endpoints identified by endpoint will be used by this pipe (see the discussion of message pipes in Chapter 5 of the USB Specification). **USB_DIR_INOUT** may be specified only for control pipes.

maxPayload specifies the largest data payload supported by this endpoint. Normally a USB device will declare the maximum payload size it supports on each endpoint in its configuration descriptors. The client will typically read these descriptors using the USB D Configuration routines, then parse the descriptors to retrieve the appropriate maximum payload value.

bandwidth specifies the bandwidth required for this pipe. For control and bulk pipes, this parameter should be zero. For interrupt pipes, this parameter should express the number of bytes per frame to be transferred. For isochronous pipes, this parameter should express the number of bytes per second to be transferred.

serviceInterval specifies the maximum latency for the pipe in milliseconds. If a pipe needs to be serviced, for example, at least every 20 milliseconds, then the *serviceInterval* value should be 20. The *serviceInterval* parameter is required only for interrupt pipes. For other types of pipes, *serviceInterval* should be zero.

If the USB D succeeds in creating the pipe it returns a pipe handle in *pPipeHandle*. The client must use the pipe handle to identify the pipe in subsequent calls to the USB D transfer routines. If there is insufficient bus bandwidth available to create the pipe (as might happen for an isochronous or interrupt pipe), then the USB D will return an error and a NULL handle in *pPipeHandle*.

RETURNS OK, or ERROR if pipe could not be create

ERRNO N/A

SEE ALSO **usbTransUnitData**

usbPipeDestroy()

NAME **usbPipeDestroy()** – Destroys a USB data transfer pipe.

SYNOPSIS `STATUS usbPipeDestroy`

```
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_PIPE_HANDLE   pipeHandle   /* pipe handle */
)
```

DESCRIPTION	This routine destroys a pipe created by calling usbPipeCreate() . The caller must pass the <i>pipeHandle</i> originally returned by usbPipeCreate() .
RETURNS	OK, or ERROR if unable to destroy the pipe.
ERRNO	N/A
SEE ALSO	usbTransUnitData

usbRootNodeIdGet()

NAME	usbRootNodeIdGet() – Returns the root node for a specific USB.
------	------------------------------------------------------------------------

SYNOPSIS	<pre>STATUS usbRootNodeIdGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ UINT16 busIndex, /* Bus index */ pUSB_NODE_ID pRootId /* bfr to receive Root Id */)</pre>
----------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

DESCRIPTION	This routine returns the node ID for the root hub of the specified USB host controller. <i>busIndex</i> is the index of the desired USB host controller. The first host controller is index zero and the last host controller's index is the total number of USB host controllers, as returned by usbBusCountGet() , minus one. <pRootId> must point to a USB_NODE_ID variable in which the node ID of the root hub will be stored.
RETURNS	OK, or ERROR if unable to get the root node ID
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbShutdown()

NAME	usbShutdown() – Shuts down the USB.
------	---------------------------------------------

SYNOPSIS	<code>STATUS usbShutdown (void)</code>
DESCRIPTION	usbShutdown() should be called once for every successful call to usbInitialize() . This function frees memory and other resources used by the USB and translation unit.
RETURNS	OK, or ERROR if a shutdown failed.
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbStatisticsGet()

NAME	usbStatisticsGet() – Retrieves USB operating statistics.
SYNOPSIS	<pre>STATUS usbStatisticsGet (USB_CLIENT_HANDLE clientHandle, /* Client handle */ USB_NODE_ID nodeId, /* Node Id of node on desired USB */ pUSB_STATS pStatistics, /* Ptr to structure to receive stats */ UINT16 statLen /* Len of bfr provided by caller */) </pre>
DESCRIPTION	<p>This routine returns operating statistics for the USB to which the specified <i>nodeId</i> is connected.</p> <p>The USB copies the current operating statistics into the <i>pStatistics</i> structure provided by the caller. This structure is defined as:</p> <pre>typedef struct usb_stats { UINT16 totalTransfersIn; UINT16 totalTransfersOut; UINT16 totalReceiveErrors; UINT16 totalTransmitErrors; } USB_STATS, *pUSB_STATS;</pre> <p>This structure may grow. To provide backwards compatibility, the client must pass the size of the USB_STATS structure it has allocated in <i>statLen</i>. The USB will copy fields into this structure only up to the <i>statLen</i> indicated by the caller.</p>
RETURNS	OK
ERRNO	N/A
SEE ALSO	usbTransUnitMisc

usbStatusGet()

NAME	usbStatusGet() – Retrieves the USB status from a source such as a device or interface and so on.
SYNOPSIS	<pre>STATUS usbStatusGet (USBD_CLIENT_HANDLE clientHandle, /* Client handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ UINT16 requestType, /* Selects device/interface/endpoint */ UINT16 index, /* Interface/endpoint index */ UINT16 bfrLen, /* length of bfr */ PUINT8 pBfr, /* bfr to receive status */ PUINT16 pActLen /* bfr to receive act len xfr'd */)</pre>
DESCRIPTION	<p>This routine retrieves the current status from the device indicated by <i>nodeId</i>. <i>requestType</i> indicates the nature of the desired status as documented for the usbFeatureClear() routine.</p> <p>The status word is returned in <i>pBfr</i>. The meaning of the status varies depending on whether it was queried from the device, an interface, or an endpoint, class-specific routine, and so on, as described in the USB Specification.</p>
RETURNS	OK, or ERROR if unable to get status
ERRNO	none
SEE ALSO	usbTransUnitStd

usbSynchFrameGet()

NAME	usbSynchFrameGet() – Returns the isochronous synchronization frame of a device.
SYNOPSIS	<pre>STATUS usbSynchFrameGet (USBD_CLIENT_HANDLE clientHandle, /* Client Handle */ USBD_NODE_ID nodeId, /* Node Id of device/hub */ UINT16 endpoint, /* Endpoint to be queried */ PUINT16 pFrameNo /* Frame number returned by device */)</pre>

DESCRIPTION	It is sometimes necessary for clients to resynchronize with devices when the two are exchanging data isochronously. This routine allows a client to query a reference frame number maintained by the device. Please refer to the USB specification for more detail. <i>nodeId</i> specifies the node to query and <i>endpoint</i> specifies the endpoint on that device. Upon return, the device's frame number for the specified endpoint is returned in <i>pFrameNo</i> .
RETURNS	OK, or ERROR if unable to retrieve the synchronization frame
ERRNO	none
SEE ALSO	usbTransUnitStd

usbTransfer()

NAME **usbTransfer()** – Initiates a transfer on a USB pipe.

SYNOPSIS

```
STATUS usbTransfer
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_PIPE_HANDLE pipeHandle, /* Pipe handle */
    pUSB_IRP pIrp /* ptr to I/O request packet */
)
```

DESCRIPTION A client uses this routine to initiate a transfer on a pipe indicated by *pipeHandle*. The transfer is described by an IRP, or I/O request packet, which must be allocated and initialized by the caller before invoking **usbTransfer()**.

The **USB_IRP** structure is defined in **usb.h** as:

```
typedef struct usb_bfr_list
{
    UINT16 pid;
    pUINT8 pBfr;
    UINT16 bfrLen;
    UINT16 actLen;
} USB_BFR_LIST;

typedef struct usb_irp
{
    LINK usbLink; /* used by USBD */
    pVOID usbPtr; /* used by USBD */
    LINK hcdLink; /* used by HCD */
    pVOID hcdPtr; /* used by HCD */
    pVOID userPtr;
    UINT16 irpLen;
    int result; /* returned by USBD/HCD */
    IRP_CALLBACK usbCallback; /* used by USBD */
}
```

```
IRP_CALLBACK userCallback;  
UINT16 dataToggle;      // filled in by USB  
UINT16 flags;  
UINT32 timeout;          // defaults to 5 seconds if zero  
UINT16 startFrame;  
UINT16 transferLen;  
UINT16 dataBlockSize;  
UINT16 bfrCount;  
USB_BFR_LIST bfrList [1];  
} USB_IRP, *pUSB_IRP;
```

The length of the **USB_IRP** structure must be stored in *irpLen* and varies depending on the number of *bfrList* elements allocated at the end of the structure. By default, the structure contains a single *bfrList* element, but clients may allocate a longer structure to accommodate a larger number of *bfrList* elements.

flags defines additional transfer options. The currently defined flags are:

USB_FLAG_SHORT_OK

Treats receive (in) data underrun as **OK**.

USB_FLAG_SHORT_FAIL

Treats receive (in) data underrun as an error.

USB_FLAG_ISO_ASAP

Start an isochronous transfer immediately.

When the USB is transferring data from a device to the host the data may underrun. That is, the device may transmit less data than anticipated by the host. This may indicate an error condition, depending on the design of the device. For many devices, the underrun is completely normal and indicates the end of the data stream from the device. For other devices, the underrun indicates a transfer failure. By default, the USB and underlying USB HCD (Host Controller Driver) treat an underrun as an end-of-data indicator and do not declare an error. If the **USB_FLAG_SHORT_FAIL** flag is set, then the USB/HCD will instead treat underrun as an error condition.

For isochronous transfers, the **USB_FLAG_ISO_ASAP** specifies that the isochronous transfer should begin as soon as possible. If **USB_FLAG_ISO_ASAP** is not specified, then *startFrame* must specify the starting frame number for the transfer. The **usbCurrentFrameGet()** routine allows a client to retrieve the current frame number and a value called the frame scheduling window for the underlying USB host controller. The frame window specifies the maximum number of frames into the future (relative to the current frame number) which may be specified by *startFrame*. *startFrame* should be specified only for isochronous transfers.

dataBlockSize may also be specified for isochronous transfers. If non-zero, *dataBlockSize* defines the granularity of the isochronous data being sent. When the underlying HCD breaks up the transfer into individual frames, it will ensure that the amount of data transferred in each frame is a multiple of this value.

timeout specifies the length of the IRP timeout in milliseconds. If the caller passes a value of zero, then the USB sets a default timeout of **USB_TIMEOUT_DEFAULT**. If no timeout is

desired, then *timeout* should be set to **USB_TIMEOUT_NONE**. Timeouts apply only to control and bulk transfers. Isochronous and interrupt transfers do not time out.

bfrList is an array of buffer descriptors which describe data buffers to be associated with this IRP. If more than the one *bfrList* element is required, then the caller must allocate the IRP by calculating the size as

```
irpLen = sizeof (USB_IRP) + (sizeof (USB_BFR_DESCR) * (bfrCount - 1))
```

transferLen must be the total length of data to be transferred. In other words, *transferLen* is the sum of all *bfrLen* entries in the *bfrList*.

pid specifies the packet type to use for the indicated buffer and is specified as **USB_PID_xxxx**.

The IRP *userCallback* routine must point to a client-supplied **IRP_CALLBACK** routine. The **usbTransfer()** routine returns as soon as the IRP has been successfully placed in a queue. If there is a failure in delivering the IRP to the HCD, then **usbTransfer()** returns an error. The result of the IRP should be checked after the *userCallback* routine has been invoked.

RETURNS	OK, or ERROR if unable to submit IRP for transfer
ERRNO	N/A
SEE ALSO	usbTransUnitData

usbTransferAbort()

NAME **usbTransferAbort()** – Aborts a transfer.

SYNOPSIS

```
STATUS usbTransferAbort
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_PIPE_HANDLE pipeHandle, /* Pipe handle */
    pUSB_IRP pIrp /* ptr to I/O to abort */
)
```

DESCRIPTION This routine aborts an IRP submitted through a call to **usbTransfer()**.

RETURNS OK, or **ERROR** if unable to abort transfer

ERRNO N/A

SEE ALSO **usbTransUnitData**

usbdVendorSpecific()

NAME **usbdVendorSpecific()** – Allows clients to issue vendor-specific USB requests.

SYNOPSIS

```
STATUS usbdVendorSpecific
(
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID        nodeId,      /* Node Id of device/hub */
    UINT8               requestType, /* bmRequestType in USB spec. */
    UINT8               request,      /* bRequest in USB spec. */
    UINT16              value,        /* wValue in USB spec. */
    UINT16              index,        /* wIndex in USB spec. */
    UINT16              length,       /* wLength in USB spec. */
    pUINT8              pBfr,         /* ptr to data buffer */
    pUINT16              pActLen       /* actual length of IN */
)
```

DESCRIPTION Certain devices may implement vendor-specific USB requests which cannot be generated using the standard routines described elsewhere. This routine allows a client to specify directly the exact parameters for a USB control pipe request.

requestType, *request*, *value*, *index*, and *length* correspond exactly to the *bmRequestType*, *bRequest*, *wValue*, *wIndex*, and *wLength* fields defined by the USB Specification. If *length* is greater than zero, then *pBfr* must be a non-NULL pointer to a data buffer which will provide or accept data, depending on the direction of the transfer.

Vendor-specific requests issued through this routine are always directed to the control pipe of the device specified by *nodeId*. This routine formats and sends a setup packet based on the parameters provided. If a non-NULL *pBfr* is also provided, then additional in or out transfers will be performed following the setup packet. The direction of these transfers is inferred from the direction bit in the *requestType* param. For in transfers, the length of the data transferred will be stored in *pActLen* if *pActLen* is not NULL.

RETURNS OK, or ERROR if unable to execute vendor-specific request

ERRNO N/A

SEE ALSO **usbTransUnitData**

usbdVersionGet()

NAME **usbdVersionGet()** – Returns USB version information.

SYNOPSIS STATUS usbdVersionGet

```
(
  pUINT16 pVersion, /* UINT16 bfr to receive version */
  pCHAR   pMfg      /* bfr to receive USBD mfg string */
)
```

DESCRIPTION	<p>This routine returns the USBD version. If <i>pVersion</i> is not NULL, the USBD returns its version in BCD in <i>pVersion</i>. For example, version 1.02 would be coded as 01h in the high byte and 02h in the low byte.</p> <p>If <i>pMfg</i> is not NULL it must point to a buffer of at least USB_D_NAME_LEN bytes in length in which the USBD will store the NULL-terminated name of the USBD manufacturer (e.g., "Wind River Systems" + \0).</p>
RETURNS	OK , or ERROR
ERRNO	none
SEE ALSO	usbTransUnitMisc

usbtuDataUrbCompleteCallback()

NAME	usbtuDataUrbCompleteCallback() – Callback called on URB completion.
SYNOPSIS	<pre>USBHST_STATUS usbtuDataUrbCompleteCallback (pUSBHST_URB urbPtr /* URB pointer */)</pre>
DESCRIPTION	This routine is called from an interrupt context by the USBD on a URB completion.
RETURNS	USBHST_SUCCESS on success or USBHST_FAILURE on failure
ERRNO	N/A
SEE ALSO	usbTransUnitData

usbtuDataVendorSpecificCallback()

NAME	usbtuDataVendorSpecificCallback() – Callback called on Vendor Specific Request
-------------	----------------------------------------------------------------------------------------

SYNOPSIS	<pre>USBHST_STATUS usbtuDataVendorSpecificCallback (pUSBHST_URB urbPtr /* URB pointer */)</pre>
DESCRIPTION	completion. This routine is called from an interrupt context by the USBBD on a vendor-specific request completion.
RETURNS	USBHST_SUCCESS
ERRNO	N/A
SEE ALSO	usbTransUnitData

usbtuInitClientIrpCompleteThreadFn()

NAME	usbtuInitClientIrpCompleteThreadFn() – Client thread routine
SYNOPSIS	<pre>VOID usbtuInitClientIrpCompleteThreadFn (pVOID driverParam)</pre>
DESCRIPTION	This routine is executed by a client thread. The thread waits in the message queue created for the client. The message is of the type USBTU_CLIENTMSG. It acts based on the USBTU_EVENTCODE in the message.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbtuInitClientThreadFn()

NAME	usbtuInitClientThreadFn() – Client thread routine
SYNOPSIS	<pre>VOID usbtuInitClientThreadFn</pre>

```
(
    pVOID driverParam
)
```

DESCRIPTION	This routine is executed by a client thread. The thread waits in the message queue created for the client. The message is of the type <code>USBTU_CLIENTMSG</code> . It acts based on the <code>USBTU_EVENTCODE</code> in the message.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbtuInitDeviceAdd()

NAME **usbtuInitDeviceAdd()** – Device attach callback

SYNOPSIS

```
USBHST_STATUS usbtuInitDeviceAdd
(
    UINT32 hDevice,
    UINT8 interfaceNumber,
    UINT8 speed,
    void** ppDriverData
)
```

DESCRIPTION	This function is called from an interrupt context by USBBD on a device attach.
RETURNS	<code>USBHST_SUCCESS</code> , or <code>USBHST_FAILURE</code> on failure
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbtuInitDeviceRemove()

NAME **usbtuInitDeviceRemove()** – Device detach callback

SYNOPSIS

```
VOID usbtuInitDeviceRemove
(
    UINT32 hDevice,
```

```
PVOID pDriverData
)
```

DESCRIPTION	This function is called from an interrupt context by USB D on a device detach.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbTUnitDeviceResume()

NAME	usbTUnitDeviceResume() – Device resume callback
SYNOPSIS	<pre>VOID usbTUnitDeviceResume (UINT32 hDevice, PVOID pSuspendData)</pre>
DESCRIPTION	This function is called from an interrupt context by USB D on a device resume.
RETURNS	N/A
ERRNO	N/A
SEE ALSO	usbTransUnitInit

usbTUnitDeviceSuspend()

NAME	usbTUnitDeviceSuspend() – Device suspend callback
SYNOPSIS	<pre>VOID usbTUnitDeviceSuspend (UINT32 hDevice, PVOID ppSuspendData)</pre>
DESCRIPTION	This function is called from the interrupt context by USB D on a device suspend.

RETURNS N/A

ERRNO N/A

SEE ALSO **usbTransUnitInit**

usbtuInitThreadFn()

NAME **usbtuInitThreadFn()** – Translation unit thread routine

SYNOPSIS

```
VOID usbtuInitThreadFn
(
    pVOID param /* User Parameter */
)
```

DESCRIPTION This routine is executed by the translation unit thread. The thread waits in the message queue created for the translation unit. The message is of the type **USBTU_TUMSG**. It performs appropriate actions based on the **USBTU_EVENTCODE** in the message.

RETURNS N/A

ERRNO N/A

SEE ALSO **usbTransUnitInit**

vxvUsbEhciRegister()

NAME **vxvUsbEhciRegister()** – registers the EHCI Controller with vxBus

SYNOPSIS

```
VOID vxvUsbEhciRegister (void)
```

DESCRIPTION This routine registers the EHCI host controller Driver and EHCI Root-hub driver with vxBus. Note that this can be called early in the initialization sequence.

RETURNS Nothing

ERRNO None.

SEE ALSO **usbEhcdInitExit**

vxUSBhciRegister()

NAME **vxUSBhciRegister()** – registers OHCI driver with vxBus

SYNOPSIS `VOID vxUSBhciRegister (void)`

DESCRIPTION This routine registers the OHCI Driver with vxBus. The registration is done for both PCI and Local bus type by calling the routine `vxDevRegister()`.

Once the OHCI driver is registered, this function also registers the OHCI Root hub as bus-controller type with vxBus

RETURNS None

ERRNO none

SEE ALSO **usbOhci**

vxUSBuhciRegister()

NAME **vxUSBuhciRegister()** – register the USB UHCI Host Controller Driver with vxBus.

SYNOPSIS `VOID vxUSBuhciRegister (void)`

DESCRIPTION This routine registers the UHCI Host Controller Driver with vxBus and can be called from either the target initialization code (bootup) or during runtime.

RETURNS None

ERRNO None.

SEE ALSO **usbUhdInitialization**