## Contents

1 Libraries ........................................................................................................... 1

2 Routines ........................................................................................................... 65
Libraries

cmdParser – Command line parser routines
device – USBD device management functionality
ossLib – O/S-independent services for vxWorks
urb – USB request block handler routines
usbBulkDevLib – USB Bulk Only Mass Storage class driver
usbCbiUfiDevLib – USB CBI Mass Storage class driver for UFI sub-class
usbDescrCopyLib – USB descriptor copy utility functions
usbEhcdBandwidth – contains the bandwidth functions of EHCD
usbEhcdEventHandler – USB EHCI HCD interrupt handler
usbEhcdInitExit – USB EHCI HCD initialization routine
usbEhcdRhEmulation – USB EHCI HCD Roothub Emulation
usbEhcdTransferManagement – transfer management functions of the EHCD
usbEhcdUtil – contains the utility functions of EHCD
usbHalDeviceControlStatus – HAL Device Control and Status handler module
usbHalEndpoint – HAL Endpoint specific functionalities
usbHalInitExit – HAL initialization and uninitialization functionalities
usbHalInterruptHandler – USB HAL interrupt handler module
usbHalUtil – Utility functions of HAL
usbHandleLib – handle utility functions
usbHubBusManager – Manages topology and other information about bus
usbHubClassInterface – functions used by USB host stack
usbHubEventHandler – Functions for handling HUB events
usbHubGlobalVariables – Stores all the global variables used by HUB class driver
usbHubInitialization – Initialization and cleanup of HUB class driver
usbHubMisc – debug functionality functions
usbHubPortEventHandler – Functions for handling the port events
usbHubUtility – Utility functions to provide the functionality of HUB class driver
usbKeyboardLib – USB keyboard class drive with vxWorks SIO interface
usbLib – USB utility functions
usbListLib – Linked list utility functions
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>usbMouseLib</td>
<td>USB mouse class drive with vxWorks SIO interface</td>
<td>22</td>
</tr>
<tr>
<td>usbOhci</td>
<td>USB OHCI Driver Entry and Exit points</td>
<td>24</td>
</tr>
<tr>
<td>usbOhciDebug</td>
<td>USB OHCI Debug Routines</td>
<td>25</td>
</tr>
<tr>
<td>usbOhciIsr</td>
<td>USB OHCI Driver Interrupt Handler</td>
<td>25</td>
</tr>
<tr>
<td>usbOhciRootHubEmulation</td>
<td>USB OHCI Root hub Emulation</td>
<td>25</td>
</tr>
<tr>
<td>usbOhciTransferManagement</td>
<td>Transfer Management of USB OHCI</td>
<td>26</td>
</tr>
<tr>
<td>usbPciStub</td>
<td>System-specific PCI Functions</td>
<td>26</td>
</tr>
<tr>
<td>usbPegasusEnd</td>
<td>USB Ethernet driver for the Pegasus USB-Ethernet adapter</td>
<td>27</td>
</tr>
<tr>
<td>usbPrinterLib</td>
<td>USB printer class drive with vxWorks SIO interface</td>
<td>29</td>
</tr>
<tr>
<td>usbQueLib</td>
<td>O/S-independent queue functions</td>
<td>31</td>
</tr>
<tr>
<td>usbSpeakerLib</td>
<td>USB speaker class drive with vxWorks SEQ_DEV interface</td>
<td>31</td>
</tr>
<tr>
<td>usbTargDefaultPipe</td>
<td>Handles the requests to the default control pipe</td>
<td>34</td>
</tr>
<tr>
<td>usbTargDeviceControl</td>
<td>modules for handling pipe specific requests</td>
<td>34</td>
</tr>
<tr>
<td>usbTargInitExit</td>
<td>USB Initialization/Uninitialization modules</td>
<td>35</td>
</tr>
<tr>
<td>usbTargKbdLib</td>
<td>USB keyboard target exerciser/demonstration</td>
<td>35</td>
</tr>
<tr>
<td>usbTargMsLib</td>
<td>Mass Storage routine library</td>
<td>36</td>
</tr>
<tr>
<td>usbTargPipeFunc</td>
<td>modules for handling pipe specific requests</td>
<td>37</td>
</tr>
<tr>
<td>usbTargPrnLib</td>
<td>USB printer target exerciser/demonstration</td>
<td>37</td>
</tr>
<tr>
<td>usbTargRbcCmd</td>
<td>Reduced Block Command set routine library</td>
<td>38</td>
</tr>
<tr>
<td>usbTargRbclLib</td>
<td>USB Reduced Block Command set routine library</td>
<td>39</td>
</tr>
<tr>
<td>usbTargUtil</td>
<td>Utility Functions</td>
<td>39</td>
</tr>
<tr>
<td>usbTcdIsp1582DeviceControl</td>
<td>Defines modules for Device Control Features</td>
<td>39</td>
</tr>
<tr>
<td>usbTcdIsp1582Endpoint</td>
<td>Endpoint Related Routines</td>
<td>40</td>
</tr>
<tr>
<td>usbTcdIsp1582InitExit</td>
<td>Initialization/uninitialization for ISP 1582 TCD</td>
<td>40</td>
</tr>
<tr>
<td>usbTcdIsp1582Interrupt</td>
<td>defines modules for interrupt handling</td>
<td>41</td>
</tr>
<tr>
<td>usbTcdIsp1582Util</td>
<td>This module contains ISP 1582 utility functions</td>
<td>41</td>
</tr>
<tr>
<td>usbTcdNET2280DeviceControl</td>
<td>defines modules for device control features</td>
<td>41</td>
</tr>
<tr>
<td>usbTcdNET2280Endpoint</td>
<td>Endpoint Related Routines</td>
<td>42</td>
</tr>
<tr>
<td>usbTcdNET2280InitExit</td>
<td>initialization/uninitialization for NET2280 TCD</td>
<td>42</td>
</tr>
<tr>
<td>usbTcdNET2280Interrupt</td>
<td>defines modules for interrupt handling</td>
<td>43</td>
</tr>
<tr>
<td>usbTcdPdiusbd12DeviceControl</td>
<td>Defines modules for Device Control Features</td>
<td>43</td>
</tr>
<tr>
<td>usbTcdPdiusbd12Endpoint</td>
<td>Endpoint related functionalities of PDIUSB12</td>
<td>43</td>
</tr>
<tr>
<td>usbTcdPdiusbd12InitExit</td>
<td>Initialization/uninitialization for PDIUSB12 TCD</td>
<td>44</td>
</tr>
<tr>
<td>usbTcdPdiusbd12Interrupt</td>
<td>Defines modules for interrupt handling</td>
<td>44</td>
</tr>
<tr>
<td>usbTcdPdiusbd12Util</td>
<td>This module contains PDIUSB12 utility functions</td>
<td>45</td>
</tr>
<tr>
<td>usbTransUnitData</td>
<td>Translation Unit Data Transfer Interfaces</td>
<td>45</td>
</tr>
<tr>
<td>usbTransUnitInit</td>
<td>Translation Unit Initialization interfaces</td>
<td>45</td>
</tr>
<tr>
<td>usbTransUnitMisc</td>
<td>translation unit miscellaneous functions</td>
<td>46</td>
</tr>
<tr>
<td>usbTransUnitStd</td>
<td>translation unit standard requests interfaces</td>
<td>47</td>
</tr>
<tr>
<td>usbUhdInitialization</td>
<td>USB UHCI HCD initialization routine</td>
<td>47</td>
</tr>
<tr>
<td>usbUhdIsr</td>
<td>USB UHCI HCD interrupt handler</td>
<td>48</td>
</tr>
<tr>
<td>usbUhdManagePort</td>
<td>USB UHCI HCD port status handler</td>
<td>48</td>
</tr>
<tr>
<td>usbUhdRhEmulate</td>
<td>USB UHCI HCD Roothub Emulation</td>
<td>48</td>
</tr>
<tr>
<td>usbUhdScheduleQSupport</td>
<td>USB UHCD HCD schedule queue support</td>
<td>49</td>
</tr>
<tr>
<td>usbUhdScheduleQWaitForSignal</td>
<td>USB UHCD HCD ISR support routines</td>
<td>49</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td><code>usbUhcdScheduleQueue</code></td>
<td>USB UHCD HCD schedule queue routines</td>
<td>50</td>
</tr>
<tr>
<td><code>usbUhcdSupport</code></td>
<td>USB UHCD HCD register access routines</td>
<td>50</td>
</tr>
<tr>
<td><code>usbd</code></td>
<td>USBD Routines</td>
<td>50</td>
</tr>
<tr>
<td><code>usbdMisc</code></td>
<td>user specific information</td>
<td>51</td>
</tr>
<tr>
<td><code>usrUsbAudioDemo</code></td>
<td>USB Audio Demo</td>
<td>51</td>
</tr>
<tr>
<td><code>usrUsbBulkDevInit</code></td>
<td>USB Mass Storage Bulk driver initialization</td>
<td>52</td>
</tr>
<tr>
<td><code>usrUsbCbiUfiDevInit</code></td>
<td>USB Mass Storage CBI driver initialization</td>
<td>53</td>
</tr>
<tr>
<td><code>usrUsbHcdEhciInit</code></td>
<td>Initialization of a EHCI Host Controller Driver</td>
<td>53</td>
</tr>
<tr>
<td><code>usrUsbHcdOhciInit</code></td>
<td>Initialization of a OHCI Host Controller Driver</td>
<td>54</td>
</tr>
<tr>
<td><code>usrUsbHcdUhciInit</code></td>
<td>Initialization of a UHCI Host Controller Driver</td>
<td>54</td>
</tr>
<tr>
<td><code>usrUsbInit</code></td>
<td>Initialization of the USB Host stack</td>
<td>54</td>
</tr>
<tr>
<td><code>usrUsbKbdInit</code></td>
<td>Initialization of the USB Keyboard driver</td>
<td>55</td>
</tr>
<tr>
<td><code>usrUsbMseInit</code></td>
<td>Initialization of the USB Mouse driver</td>
<td>55</td>
</tr>
<tr>
<td><code>usrUsbPciInit</code></td>
<td>USB Host Controller host bus specific initialization</td>
<td>56</td>
</tr>
<tr>
<td><code>usrUsbPegasusEndInit</code></td>
<td>Initialization of the USB END driver</td>
<td>59</td>
</tr>
<tr>
<td><code>usrUsbPrnInit</code></td>
<td>Initialization of the USB Printer driver</td>
<td>59</td>
</tr>
<tr>
<td><code>usrUsbSpkrInit</code></td>
<td>Initialization of the USB Speaker driver</td>
<td>60</td>
</tr>
<tr>
<td><code>usrUsbTargInit</code></td>
<td>WindView Instrumentation function Definition</td>
<td>60</td>
</tr>
<tr>
<td><code>usrUsbTargKbdInit</code></td>
<td>Initialization of the keyboard function driver</td>
<td>61</td>
</tr>
<tr>
<td><code>usrUsbTargMslInit</code></td>
<td>Initialization of the keyboard function driver</td>
<td>61</td>
</tr>
<tr>
<td><code>usrUsbTargPciInit</code></td>
<td>Initialization of USB Target Controller PCI Interfaces</td>
<td>61</td>
</tr>
<tr>
<td><code>usrUsbTargPrnInit</code></td>
<td>Initialization of the printer function driver</td>
<td>62</td>
</tr>
<tr>
<td><code>usrUsbTool</code></td>
<td>USB Exerciser</td>
<td>62</td>
</tr>
</tbody>
</table>
1 Libraries
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.

```c
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).

**device**

**NAME**

device – USBD device management functionality

**ROUTINES**

- `usbHstSelectiveSuspend()` – suspend device.
- `usbHstSelectiveResume()` – resume device.
- `usbHstResetDevice()` – called by the class drivers to reset its device

**DESCRIPTION**

Device manager provides for handling Plug & Play and Power Management. The Device Manager provides the interfaces for the following functionality:

1. Handle device connection and disconnection.
2. Configure a newly attached device.
3. Handle device suspend and resume.
4. Interfaces for the client software to selectively suspend or resume a device.

This file is included by the `usbd.c` source file.

**INCLUDE FILES**

none

---

**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 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 old method of USB malloc and free.
ossTime() – Returns 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

urb

NAME
urb – USB request block handler routines

ROUTINES
usbHstGetDescriptor() – get USB Descriptor
usbHstGetStatus() – Get USB Status
usbHstClearFeature() – clear feature USB request
usbHstGetConfiguration() – get configuration USB request
usbHstGetInterface() – get interface USB request
usbHstSetConfiguration() – set configuration USB request
usbHstSetFeature() – set feature USB request
usbHstSetDescriptor() – set descriptor USB request
usbHstSetInterface() – set interface USB request
usbHstSetSynchFrame() – set synch frame USB request
usbHstSetBitRate() – set USB bit rate
usbHstURBSSubmit() – submit USB request block
usbHstURBCancel() – cancel USB Request Block
usbHstGetFrameNumber() – get USB frame number
This module provides interfaces to communicate with the USB Device. The class drivers can use these interfaces to:
1. Issue USB Standard Requests to the device.
2. Issue Class Specific or Vendor Specific using data transfers on the Default Control Endpoint.
3. Perform all data transfers to the device.
4. Cancel data transfers.

This file is included by the `usbd.c` source file.

**INCLUDE FILES**
none

---

**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.
- `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 (USBD). All interaction with the USB buses and devices is handled through the USBD.

**INITIALIZATION**
The class driver must be initialized with `usbBulkDevInit()`. It is assumed that USBD is already initialized and attached to atleast one USB Host Controller. `usbBulkDevInit()` registers the class driver as a client module with USBD. 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 with in 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 USBD 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**


---

**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.
**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. The driver implements external APIs which would be expected of a standard block device driver.

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.

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.

**INITIALISATION**

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 atleast 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.

**DATA FLOW**

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.

**OTHER FUNCTIONS**

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.
NOTE
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.

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.

INCLUDE FILES
`usbCbiUfiDevLib.h`, `blkIo.h`

SEE ALSO
`USB Mass Storage Class - Control/Bulk/Interrupt Transport Specification Revision 1.0`, `USB Mass Storage Class - UFI Command specification Revision 1.0`

---

**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**
`USB specification, revision 2.0`, `EHCI specification, revision 1.0`
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

usbEhcdInit() – initializes the EHCI Host Controller
usbEhcdExit() – uninitializes the EHCI Host Controller

DESCRIPTION
This contains the initialization and uninitialization functions provided by the EHCI Host Controller Driver.

INCLUDE FILES

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
1 Libraries

**usbEhcdTransferManagement**

**NAME**

`usbEhcdTransferManagement` – transfer management functions of the EHCD

**ROUTINES**

**DESCRIPTION**

This module defines the interfaces which are registered with the USBD 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**

Implements a set of general-purpose handle creation and validation functions.

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."

**INCLUDE FILES**

`usbHandleLib.h`

usbHubBusManager

**NAME**

`usbHubBusManager` – Manages topology and other information about bus

**ROUTINES**

**DESCRIPTION**

This is the module that manages the topology and other information about a bus. This is an autonomous thread that is initiated every time a root hub is detected and this ends only when the USB Hub Class Driver is unloaded or when the root hub is disabled. This is used in conjunction with the Port Event Handling Module and Hub Event Handling Module.

**INCLUDE FILES**

`usb2/usbOsal.h`, `usb2/usbHst.h`, `usb2/usbHubCommon.h`, `usb2/usbHubUtility.h`, `usb2/usbHubEventHandler.h`, `usb2/usbHubPortEventHandler.h`, `usb2/usbHubBusManager.h`, `usb2/usbHubGlobalVariables.h`
usbHubClassInterface

NAME

usbHubClassInterface – functions used by USB host stack

ROUTINES

DESCRIPTION

This module implements the functions to be used by the USB Host Software Stack. The following functions are described in this module:

- Function for plug and play.
- Function for power management.
- Function for root hub management.
- Function for selective suspend and resume.
- Function for checking hub power capability.
- Function for resetting a device.

INCLUDE FILES

usb2/usbOsal.h, usb2/usbHubCommon.h, usb2/usbHubGlobalVariables.h, usb2/usbHubUtility.h, usb2/usbHubClassInterface.h

usbHubEventHandler

NAME

usbHubEventHandler – Functions for handling HUB events

ROUTINES

DESCRIPTION

This provides the functions for handling the hub events. This is used in conjunction with the Port Event Handling Module and Bus Manager Module.

INCLUDE FILES

usb2/usbOsal.h, usb2/usbHst.h, usb2/usbHubCommon.h, usb2/usbHubUtility.h, usb2/usbHubEventHandler.h
### usbHubGlobalVariables

**NAME**

usbHubGlobalVariables – Stores all the global variables used by HUB class driver

**ROUTINES**

This module stores all the global variables used by HUB class driver.

**INCLUDE FILES**

usb2/usbOsal.h usb2/usbHubCommon.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/usbOsal.h, usb2/usbHubCommon.h, usb2/usbHubGlobalVariables.h, usb2/usbHubInitialization.h, usb2/usbHubClassInterface.h, usb2/usbHst.h, usb2/usbHcdInstr.h

### usbHubMisc

**NAME**

usbHubMisc – debug functionality functions

**ROUTINES**

**DESCRIPTION**

This module provides the debug functionality functions for the USB Hub Class Driver

**INCLUDE FILES**

usb2/usbHubCommon.h, usb2/usbHubUtility.h, usb2/usbHubMisc.h
usbHubPortEventHandler

NAME  
_usbHubPortEventHandler – Functions for handling the port events

ROUTINES

DESCRIPTION  
This provides the functions for handling the port events. This module is used in conjunction with the Hub Event Handling Module and Bus Manager Module.

INCLUDE FILES  
usb2/usbOsal.h, usb2/usbHst.h, usb2/usbHubGlobalVariables.h, usb2/usbHubUtility.h, usb2/usbHubPortEventHandler.h, usb2/usbHubEventHandler.h

usbHubUtility

NAME  
_usbHubUtility – Utility functions to provide the functionality of HUB class driver

ROUTINES

DESCRIPTION  
This module provides the utility functions required for the functioning of the USB Hub Class Driver.

INCLUDE FILES  
usb2/usbOsal.h, usb2/usbHst.h, usb2/usbHubUtility.h, usb2/usbHubGlobalVariables.h, usb2/usbHubBusManager.h, usb2/usbHubEventHandler.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 \texttt{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 (USBD). All interaction with the USB buses and devices is handled through the USBD.

\textbf{INITIALIZATION}

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

Prior to calling \texttt{usbKeyboardDevInit()}, the caller must ensure that the USBD has been properly initialized by calling - at a minimum - \texttt{usbdInitialize()}. It is also the caller's responsibility to ensure that at least one USB HCD (USB Host Controller Driver) is attached to the USBD - using the USBD function \texttt{usbdHcdAttach()} - before keyboard operation can begin. However, it is not necessary for \texttt{usbdHcdAttach()} to be called prior to initializing \texttt{usbKeyboardLib.c}. \texttt{usbKeyboardLib.c} uses the USBD 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 USBD at run time - as may be required, for example, in systems supporting hot swapping of hardware.

\texttt{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 USBD and USB HCD(s), so there is no need for a caller (or BSP) to connect interrupts on behalf of \texttt{usbKeyboardLib.c}. For the same reason, there is no post-interrupt-connect initialization code and \texttt{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 USBD and USB HCD always operate in an interrupt mode.

TYPOGRAPHIC REPEAT

USB keyboards do not implement typematic 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 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 number of device configurations
- `usbConfigDescrGet()` – reads 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() – Add an element to a linked list
- usbListLinkProt() – Add an element to a list guarded by a mutex
- usbListUnlink() – Remove 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 next pStruct in a linked list

**DESCRIPTION**
This file implements a set of general-purpose linked-list functions which are portable across OS’s. Linked lists are a collection of LINK structures. Each LINK structure contains a forward a backward list pointer. Each LINK structure also contains a pStruct field which points (typically) to the caller’s structure which contains the LINK structure.

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, and the client’s structure may even contain more than one LINK field - allowing the structure to be linked into multiple lists simultaneously.

usbListFirst() retrieves the first structure on a linked list and usbListNext() retrieves subsequent structures.

**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
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 mouses 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 mouses 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 (USBD). All interaction with the USB buses and devices is handled through the USBD.

INITIALIZATION
As with standard SIO drivers, this driver must be initialized by calling usbMouseDevInit(). usbMouseDevInit() in turn initializes its connection to the USBD 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 USBD 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 USBD - using the USBD function usbdHcdAttach() - before mouse operation can begin. However, it is not necessary for usbdHcdAttach() to be called prior to initializing usbMouseLib.c. usbMouseLib.c uses the USBD 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 USBD 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 USBD 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 usbKeboardLib.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:

```c
typedef STATUS (*REPORT_CALLBACK)(
    void *arg,
    pHID_MOUSE_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

usbOhciInit() – initialise the USB OHCI Host Controller Driver.
usbOhciExit() – uninitialise the USB OHCI Host Controller Driver.

DESCRIPTION

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

INCLUDE FILES

usb2/usbOhci.h, usb2/usbOhciRegisterInfo.h, usb2/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 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 used to debug the OHCI driver.

INCLUDE FILES

usbOhci.h, usbOhciRegisterInfo.h, usbOhciTransferManagement.h

usbOhciIsr

NAME

usbOhciIsr – USB OHCI Driver Interrupt Handler

ROUTINES

DESCRIPTION

This file defines the interrupt handler for the USB OHCI Driver.

INCLUDE FILES

usbOhci.h, usbOhciRegisterInfo.h

usbOhciRootHubEmulation

NAME

usbOhciRootHubEmulation – USB OHCI Root hub Emulation

ROUTINES

DESCRIPTION

This file defines the functions for implementing root hub emulation

INCLUDE FILES

usbOhci.h, usbOhciRegisterInfo.h, usbOhciTransferManagement.h
### usbOhciTransferManagement

**NAME**
usbOhciTransferManagement – Transfer Management of USB OHCI

**ROUTINES**
- BufferPageCross() – checks if there is page cross in transfer buffer

**DESCRIPTION**
This file defines the transfer management of the USB OHCI driver.

**INCLUDE FILES**
usbOhci.h, usbOhciRegisterInfo.h, usbOhciTransferManagement.h

### usbPciStub

**NAME**
usbPciStub – System-specific PCI Functions

**ROUTINES**
- usbPciClassFind() – Locates PCI devices by class.
- usbPciByteGet() – Returns a UINT8 configuration value
- usbPciWordGet() – Returns a UINT16 configuration value
- usbPciDwordGet() – Returns a UINT32 configuration value
- usbPciConfigHeaderGet() – Reads a device’s PCI configuration header
- usbPciByteIn() – input a byte from PCI I/O space
- usbPciWordIn() – input a word from PCI I/O space
- usbPciDwordIn() – input a dword from PCI I/O space
- usbPciByteOut() – output a byte to PCI I/O space
- usbPciWordOut() – output a word to PCI I/O space
- usbPciDwordOut() – outputs a dword to PCI I/O space
- usbPciMemioOffset() – Return PCI MEMIO to CPU MEMIO offset
- usbMemToPci() – Convert a memory address to a PCI-reachable memory address
- usbPciToMem() – Convert a PCI-reachable address to a CPU-reachable pointer
- usbPciMemInvalidate() – Invalidate cache for a region of memory
- usbPciMemFlush() – Flush a region of memory through the cache
- usbPciIntConnect() – Connect to a interrupt vector
- usbPciIntDisconnect() – Removes an interrupt handler

**DESCRIPTION**
This file defines a skeleton of functions to be used for accessing the PCI bus capabilities. These functions allow PCI device drivers to be written independent of the underlying O/S’s PCI access mechanisms.

The name of each function in this group begins with "usb" to represent "Device Driver Services."

**INCLUDE FILES**
none
**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 USBD. 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 USBD. 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, findout 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 findout 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**

drv/usb/usbPegasusEnd.h
end.h, endLib.h, lstLib.h, etherMultiLib.h, usb/usbPlatform.h, usb/usb.h,usb/usblistLib.h, usb/usbdlLib.h, usb/usblib.h, drv/usb/usbPegasusEnd.h
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 (USBD). All interaction with the USB buses and devices is handled through the USBD.

INITIALIZATION

As with standard SIO drivers, this driver must be initialized by calling usbPrinterDevInit(). usbPrinterDevInit() in turn initializes its connection to the USBD 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 USBD 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 USBD - using the USBD function `usbdHcdAttach()` - before printer operation can begin. However, it is not necessary for `usbdHcdAttach()` to be called prior to initializing `usbPrinterLib.c`. `usbPrinterLib.c` uses the USBD 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 USBD 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 USBD 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 a O/S-independent queue structure  
- usbQueueDestroy() – Destroys a queue  
- usbQueuePut() – Puts a message onto a queue  
- usbQueueGet() – Retrieves a message from a queue

DESCRIPTION  This file contains a generic implementation of O/S-independent queue functions which are built on top of the the ossLib library’s mutex and semaphore functions.

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 on a queue is described by a USB_QUEUE structure which contains msg, wParam, and lParam fields. The values of these fields are completely arbitrary and may be used in any way desired 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  This module implements the class driver for USB speaker devices. USB speakers 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.

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 speakers. As with other VxWorks USB class drivers, the standard driver interface has been expanded to support features unique to the USB and to speakers in general. 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.

**INITIALIZE**

As with standard SEQ_DEV drivers, this driver must be initialized by calling `usbSpeakerDevInit()`. `usbSpeakerDevInit()` in turn initializes its connection to the USBD 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()`.

Prior to calling `usbSpeakerDevInit()`, the caller must ensure that the USBD 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 USBD - using the USB function `usbdHcdAttach()` - before speaker operation can begin. However, it is not necessary for `usbdHcdAttach()` to be called prior to initializing `usbSpeakerLib`. `usbSpeakerLib.c` uses the USB dynamic attach services and is capable of recognizing USB speaker attachment and removal on the fly. Therefore, it is possible for USB HCDs to be attached to or detached from the USBD at run time - as may be required, for example, in systems supporting hot swapping of hardware.

**RECOGNIZING & HANDLING USB SPEAKERS**

As noted earlier, the operation of USB speakers is defined in the USB Audio Class Specification. Speakers, loosely defined, are those USB audio devices which provide an "Output Terminal". For each USB audio device, `usbSpeakerLib` examines the descriptors which enumerate the "units" and "terminals" contained within the device. These descriptors define 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 USBD'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 speaker 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.

DATA FLOW

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.

IOCTLs

usbSpeakerLib implements a number of IOCTLs unique to the handling of audio data and devices. usbSpeakerLib provides IOCTLs to set the following controls: mute, volume, bass, mid-range, and treble. usbSpeakerLib also provides IOCTLs to be used by callers to interrogate a speaker’s audio format capabilities or to specify the audio format for a subsequent data stream.

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
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.

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.

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.

INCLUDE FILES
usb/platform.h, usb/ossLib.h, usb/usb.h, usb/listLib.h, usb/handleLib.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
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.
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`.

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.

**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**

- `usbMsCBWGet()` – get the last mass storage CBW received
- `usbMsCBWInit()` – initialize the mass storage CBW
- `usbMsCSWGet()` – get the current CSW
- `usbMsCSWInit()` – initialize the CSW
- `usbMsBulkInStall()` – stall the bulk-in pipe
- `usbMsBulkInUnStall()` – unstall the bulk-in pipe
- `usbMsBulkOutStall()` – stall the bulk-out pipe
- `usbMsBulkOutUnStall()` – unstall the bulk-out pipe
- `usbMsBulkCallbackInfo()` – returns `usbTargPrnLib` callback table
- `usbMsBulkInErpInit()` – initialize the bulk-in ERP
- `usbMsBulkOutErpInit()` – initialize the bulk-Out ERP
- `usbMsIsConfigured()` – test if the device is configured
- `usbMsBulkInErpInUseFlagGet()` – get the Bulk-in ERP inuse flag
- `usbMsBulkOutErpInUseFlagGet()` – get the Bulk-Out ERP inuse flag
- `usbMsBulkInErpInUseFlagSet()` – set the Bulk-In ERP inuse flag
- `usbMsBulkOutErpInUseFlagSet()` – set the Bulk-Out ERP inuse flag
- `usbMsTestTxCallback()` – invoked after test data transmitted
- `usbMsTestRxCallback()` – invoked after test data is received

**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**

`vxWorks.h`, `stdio.h`, `errnoLib.h`, `logLib.h`, `string.h`, `blkIo.h`, `usb/usbPlatform.h`, `usb/usb.h`, `usb/usbDescrCopyLib.h`, `usb/usbLib.h`, `usb/target/usbTargLib.h`
usbTargPipeFunc

NAME
usbTargPipeFunc – modules for handling pipe specific requests

ROUTINES
usbTargPipeCreate( ) – creates a pipe for communication on an endpoint
usbTargPipeDestroy( ) – destroys an endpoint pipe
usbTargTransfer( ) – to transfer data through a pipe
usbTargTransferAbort( ) – cancels a previously submitted USB_ERP
usbTargPipeStatusSet( ) – sets pipe stalled/unstalled status
usbTargPipeStatusGet( ) – returns the endpoint status

DESCRIPTION
This module provides interfaces for handling the various pipe specific requests.
It provides interfaces for creating and destroying pipes, submit and cancel ERPs and to get and set the pipe status information.

INCLUDE FILES
usb/usbPlatform.h, string.h, usb/ossLib.h, usb/handleLib.h,
usb/target/HalLib.h, usb/target/usbHalCommon.h, usb/target/usbTargLib.h,
usb/target/usbTargUtil.h, usb/target/usbPeriphInstr.h

usbTargPrnLib

NAME
usbTargPrnLib – USB printer target exerciser/demonstration

ROUTINES
usbTargPrnCallbackInfo( ) – returns usbTargPrnLib callback table
usbTargPrnDataInfo( ) – returns buffer status/info
usbTargPrnDataRestart() – restarts listening ERP

DESCRIPTION
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.
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 \texttt{usbTargPrnLib} or to shut it down. It performs all of its operations in response to callbacks from \texttt{usbTargLib}.

This module also exports a function, \texttt{usbTargPrnBfrInfo()}, which allows a test application to retrieve the current status of the bulk output buffer.

\section*{INCLUDE FILES}

\begin{itemize}
\item \texttt{usb/usblPlatform.h}
\item \texttt{string.h}
\item \texttt{usb/usblPrinte.h}
\item \texttt{usb/usbDescrCopyLib.h}
\item \texttt{usb/target/usblTargLib.h}
\item \texttt{drv/usbl/target/usblTargPrnLib.h}
\item \texttt{usb/target/usblHalCommon.h}
\end{itemize}

\section*{usbTargRbcCmd}

\subsection*{NAME}

\texttt{usbTargRbcCmd} – Reduced Block Command set routine library

\subsection*{ROUTINES}

\begin{itemize}
\item \texttt{usbTargRbcRead()} – read data from the RBC device
\item \texttt{usbTargRbcCapacityRead()} – read the capacity of the RBC device
\item \texttt{usbTargRbcStartStop()} – start or stop the RBC device
\item \texttt{usbTargRbcPreventAllowRemoval()} – prevent or allow the removal of the RBC device
\item \texttt{usbTargRbcVerify()} – verify the last data written to the RBC device
\item \texttt{usbTargRbcWrite()} – write to the RBC device
\item \texttt{usbTargRbcInquiry()} – retrieve inquiry data from the RBC device
\item \texttt{usbTargRbcModeSelect()} – select the mode parameter page of the RBC device
\item \texttt{usbTargRbcModeSense()} – retrieve sense data from the RBC device
\item \texttt{usbTargRbcTestUnitReady()} – test if the RBC device is ready
\item \texttt{usbTargRbcBufferWrite()} – write micro-code to the RBC device
\item \texttt{usbTargRbcFormat()} – format the RBC device
\item \texttt{usbTargRbcPersistentReserveIn()} – send reserve data to the host
\item \texttt{usbTargRbcPersistentReserveOut()} – reserve resources on the RBC device
\item \texttt{usbTargRbcRelease()} – release a resource on the RBC device
\item \texttt{usbTargRbcRequestSense()} – request sense data from the RBC device
\item \texttt{usbTargRbcReserve()} – reserve a resource on the RBC device
\item \texttt{usbTargRbcCacheSync()} – synchronize the cache of the RBC device
\item \texttt{usbTargRbcBlockDevGet()} – return opaque pointer to the RBC BLK I/O DEV device
\item \texttt{usbTargRbcBlockDevSet()} – set the pointer to the RBC BLK I/O DEV device structure.
\item \texttt{usbTargRbcBlockDevCreate()} – create an RBC BLK_DEV device.
\item \texttt{usbTargRbcVendorSpecific()} – vendor specific call
\end{itemize}

\subsection*{DESCRIPTION}

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).

\subsection*{INCLUDES}

\begin{itemize}
\item \texttt{vxWorks.h}
\item \texttt{disFsLib.h}
\item \texttt{dcacheCbio.h}
\item \texttt{ramDrv.h}
\item \texttt{usrFdiskPartLib.h}
\item \texttt{usb/usblPlatform.h}
\item \texttt{usb/usblPrinte.h}
\item \texttt{usb/target/usblTargLib.h}
\item \texttt{drv/usbl/target/usblTargMsLib.h}
\item \texttt{drv/usbl/target/usblTargRbcCmd.h}
\end{itemize}
### usbTargRbcLib

**NAME**

`usbTargRbcLib` – USB Reduced Block Command set routine library

**ROUTINES**

- `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

**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`

### usbTargUtil

**NAME**

`usbTargUtil` – Utility Functions

**ROUTINES**

**DESCRIPTION**

This file consists of utility functions which are used by the `usbTarget` Library files.

**INCLUDE FILES**

- `usb/usbdLib.h`
- `string.h`
- `usb/ossLib.h`
- `usb/usbdLib.h`
- `usb/target/HalLib.h`
- `usb/target/usbTargLib.h`
- `usb/target/usbTargUtil.h`

### usbTcdIsp1582DeviceControl

**NAME**

`usbTcdIsp1582DeviceControl` – Defines modules for Device Control Features

**ROUTINES**

**DESCRIPTION**

This module implements the hardware dependent device control and status functionalities of the TCD.
INCLUDE FILES  
usb/usbPlatform.h, usb/ossLib.h, usb/usbPciLib.h, usb/usb.h,  
usb/target/usbHalCommon.h, usb/target/usbTcd.h, drv/usb/target/usbIsp1582Eval.h,  
drv/usb/target/usbIsp1582Tcd.h, usb/target/usbPeriphInstr.h

---

**usbTcdIsp1582Endpoint**

**NAME**  
usbTcdIsp1582Endpoint – Endpoint Related Routines

**ROUTINES**

**DESCRIPTION**
This file implements the endpoint related functionalities of TCD (Target Controller Driver) for the Philips ISP 1582.

**INCLUDE FILES**
usb/usbPlatform.h, usb/ossLib.h, usb/usbPciLib.h, usb/usb.h, string.h,  
usb/target/usbHalCommon.h, usb/target/usbTcd.h, drv/usb/target/usbIsp1582Eval.h,  
drv/usb/target/usbIsp1582Tcd.h

---

**usbTcdIsp1582InitExit**

**NAME**  
usbTcdIsp1582InitExit – Initialization/uninitialization for ISP 1582 TCD

**ROUTINES**

**DESCRIPTION**
This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Philips ISP 1582.

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.

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.

**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/usbIsp1582EvalLib.h, driv/usb/target/usbIsp1582Tcd.h,  
drv/usb/target/usbIsp1582Debug.h, rebootLib.h, usb/target/usbPeriphInstr.h
**usbTcdIsp1582Interrupt**

**NAME**  
*usbTcdIsp1582Interrupt* – defines modules for interrupt handling

**ROUTINES**

**DESCRIPTION**  
This file implements the interrupt related functionalities of TCD (Target Controller Driver) for the Philips ISP 1582.

**INCLUDE FILES**  

**usbTcdIsp1582Util**

**NAME**  
*usbTcdIsp1582Util* – This module contains ISP 1582 utility functions.

**ROUTINES**

**DESCRIPTION**  
Defines all ISP 1582 utility functions. These utility functions will be used by the ISP 1582 functions to carry out 8-bits, 16-bits & 32-bits read/write access into the registers.

**INCLUDE FILES**  
usb/usbPlatform.h, usb/ossLib.h, drv/usb/target/usbIsp1582.h, drv/usb/target/usbIsp1582Eval.h

**usbTcdNET2280DeviceControl**

**NAME**  
*usbTcdNET2280DeviceControl* – defines modules for device control features

**ROUTINES**

**DESCRIPTION**  
This module implements the hardware dependent device control and status functionalities of the NET2280.

**INCLUDE FILES**  
usb/usbPlatform.h, usb/ossLib.h, usb/usb.h, usb/target/usbHalCommon.h, usb/target/usbTcd.h, drv/usb/target/usbNET2280.h, drv/usb/target/usbNET2280Tcd.h, usb/target/usbPeriphInstr.h
usbTcdNet2280Endpoint

NAME  
usbTcdNet2280Endpoint – Endpoint Related Routines

ROUTINES

DESCRIPTION  
This file implements the endpoint related functionalities of TCD (Target Controller Driver) for the Netchip NET2280

INCLUDE FILES  

usbTcdNet2280InitExit

NAME  
usbTcdNet2280InitExit – initialization/uninitialization for NET2280 TCD

ROUTINES  
usbTcdNet2280Exec() – single Entry Point for NETCHIP 2280 TCD

DESCRIPTION  
This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Netchip NET2280.

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.

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.

INCLUDE FILES  
usbTcdNET2280Interrupt

NAME
usbTcdNET2280Interrupt – defines modules for interrupt handling

ROUTINES

DESCRIPTION
This file implements the interrupt related functionalities of TCD (Target Controller Driver) for the Netchip NET2280

INCLUDE FILES

usbTcdPdiusbd12DeviceControl

NAME
usbTcdPdiusbd12DeviceControl – Defines modules for Device Control Features

ROUTINES

DESCRIPTION
This module implements the hardware dependent device control and status functionalities of the TCD.

INCLUDE FILES

usbTcdPdiusbd12Endpoint

NAME
usbTcdPdiusbd12Endpoint – Endpoint related functionalities of PDIUSBD12

ROUTINES

DESCRIPTION
This file implements the endpoint related functionalities of TCD (Target Controller Driver) for the Philips PDIUSBD12.

INCLUDE FILES
usbTcdPdiusbd12InitExit

NAME

usbTcdPdiusbd12InitExit – Initialization/uninitialization for PDIUSB12 TCD

ROUTINES

usbTcdPdiusbd12EvalExec() – single entry point for PDIUSB12 TCD

DESCRIPTION

This file implements the initialization and uninitialization modules of TCD (Target Controller Driver) for the Philips PDIUSB12.

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.

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.

INCLUDE FILES

usb/usbPlatform.h, usb/ossLib.h, usb/target/usbIsaLib.h,
drv/usb/target/usbPdiusbd12Eval.h, drv/usb/target/drvTcdPdiusbd12EvalLib.h,
drv/usb/target/usbPdiusbd12Tcd.h, drv/usb/target/usbPdiusbd12Debug.h,
usb/target/usbPeriphInstr.h

usbTcdPdiusbd12Interrupt

NAME

usbTcdPdiusbd12Interrupt – Defines modules for interrupt handling

ROUTINES

DESCRIPTION

This module implements the various hardware dependent features of PDIUSB12 which are related to interrupt handling.

INCLUDE FILES

usb/usbPlatform.h, usb/ossLib.h, usb/target/usbIsaLib.h,
drv/usb/target/usbPdiusbd12Eval.h, drv/usb/target/drvTcdPdiusbd12EvalLib.h,
drv/usb/target/usbPdiusbd12Tcd.h, drv/usb/target/usbPdiusbd12Debug.h,
usb/target/usbPeriphInstr.h
## 1 Libraries

### usbTcdPdiusbd12Util

<table>
<thead>
<tr>
<th>NAME</th>
<th>usbTcdPdiusbd12Util – This module contains PDIUSB12 utility functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINES</td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Defines all D12 utility functions. These utility functions will be used by the PDIUSB12 functions implementing the various functions codes.</td>
</tr>
<tr>
<td>INCLUDE FILES</td>
<td>usb/usbPlatform.h, usb/ossLib.h, usb/target/usbIsaLib.h, drv/target/usbPdiusbd12Eval.h, drv/target/usbTcdPdiusbd12EvalLib.h, drv/target/usbPdiusbd12Tcd.h, drv/target/usbPdiusbd12Debug.h</td>
</tr>
</tbody>
</table>

### usbTransUnitData

<table>
<thead>
<tr>
<th>NAME</th>
<th>usbTransUnitData – Translation Unit Data Transfer Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINES</td>
<td>usbdPipeCreate() – Creates a USB pipe for subsequent transfers, usbdPipeDestroy() – Destroys a USB data transfer pipe, usbdTransfer() – Initiates a transfer on a USB pipe, usbdTransferAbort() – Aborts a transfer, usbdVendorSpecific() – Allows clients to issue vendor-specific USB requests, usbtuDataUrbCompleteCallback() – Callback called on URB completion, usbtuDataVendorSpecificCallback() – Callback called on Vendor Specific Request</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Implements the Translation Unit Data Transfer Interfaces.</td>
</tr>
<tr>
<td>INCLUDE FILES</td>
<td>usbTransUnit.h, usbHcdInstr.h</td>
</tr>
</tbody>
</table>

### usbTransUnitInit

<table>
<thead>
<tr>
<th>NAME</th>
<th>usbTransUnitInit – Translation Unit Initialization interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUTINES</td>
<td>usbdInitialize() – Initialize the USBD, usbdShutdown() – Shuts down the USBD, usbdClientRegister() – Registers a new client with the USBD, usbdClientUnregister() – Unregisters a USBD client, usbdMngmtCallbackSet() – sets management callback for a client</td>
</tr>
</tbody>
</table>

---

45
usbdBusStateSet() – Sets bus state (e.g., suspend/resume)
usbdDynamicAttachRegister() – Registers client for dynamic attach notification
usbdDynamicAttachUnRegister() – Unregisters client for attach notification
usbtuInitThreadFn() – Translation Unit Thread Function
usbtuInitClientThreadFn() – Client Thread Function
usbtuInitClientIrpCompleteThreadFn() – Client Thread Function
usbtuInitDeviceAdd() – Device Attach Callback
usbtuInitDeviceRemove() – Device Detach Callback
usbtuInitDeviceSuspend() – Device Suspend Callback
usbtuInitDeviceResume() – Device Resume Callback

DESCRIPTION

Implements the Translation Unit Initialization interfaces.

In order to use the USBD, it is first necessary to invoke usbdInitialize(). Multiple calls to usbdInitialize() may be nested so long as a corresponding number of calls to usbdShutdown() are also made. This allows multiple USBD clients to be written independently and without concern for coordinating the initialization of the independent clients.

Normal USBD clients register with the USBD by calling usbdClientRegister(). 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 the real-time response for all clients.

After a client has registered, it will most often also register for dynamic attachment notification using usbdDynamicAttachRegister(). This function allows a special client callback routine to be invoked each time a USB device is attached 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

usbdHcdAttach() – attaches an HCD to the USBD
usbdHcdDetach() – Detaches an HCD from the USBD
usbdBusCountGet() – get number of USBs attached to the host.
usbdRootNodeIDGet() – returns root node for a specific USB
usbdHubPortCountGet() – returns number of ports connected to a hub
usbdNodeIDGet() – gets the id of the node connected to a hub port
usbdAddressGet() – gets the USB address for a given device
usbdAddressSet() – sets the USB address for a given device
usbTransUnitStd

**NAME**

usbTransUnitStd – translation unit standard requests interfaces

**ROUTINES**

- usbdFeatureClear() – clears a USB feature
- usbdFeatureSet() – sets a USB feature
- usbdConfigurationGet() – gets USB configuration for a device
- usbdConfigurationSet() – sets USB configuration for a device
- usbdDescriptorGet() – retrieves a USB descriptor
- usbdDescriptorSet() – sets a USB descriptor
- usbdInterfaceGet() – retrieves a device’s current interface
- usbdInterfaceSet() – sets a device’s current interface
- usbdStatusGet() – Retrieves USB status from a device/interface/etc.
- usbdSynchFrameGet() – Returns a device’s isochronous synch. frame

**DESCRIPTION**

Implements the Translation Unit Standard Requests Interfaces.

**INCLUDE FILES**

drv/usb/usbTransUnit.h, usb/pciConstants.h, usb2/usbHcdInstr.h

usbUhcdInitialization

**NAME**

usbUhcdInitialization – USB UHCI HCD initialization routine

**ROUTINES**

- usbUhcdInit() – initialise the USB UHCD Host Controller Driver.
- usbUhcdExit() – uninitialise the USB UHCD Host Controller Driver.

**DESCRIPTION**

This file contains entry and exit points for the UHCI driver.
**usbUhcdIsr**

**NAME**

`usbUhcdIsr` – 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/usbUhcdSupport.h`, `usb2/usbUhcdCommon.h`, `usb2/usbUhcdScheduleQueue.h`, `usb2/usbUhcdScheduleQSupport.h`, `usb2/BusAbstractionLayer.h`, `usbPciLib.h`, `rebootLib.h`

---

**usbUhcdManagePort**

**NAME**

`usbUhcdManagePort` – 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/usbUhcdSupport.h`, `usb2/usbUhcdCommon.h`, `usb2/usbUhcdScheduleQueue.h`, `usb2/usbUhcdScheduleQSupport.h`, `usb2/BusAbstractionLayer.h`

---

**usbUhcdRhEmulate**

**NAME**

`usbUhcdRhEmulate` – 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.
**1 Libraries**

**usbUhcdScheduleQSupport**

**NAME**

`usbUhcdScheduleQSupport` – 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/usbOsal.h`, `usb2/usbHst.h`, `usb2/BusAbstractionLayer.h`, `usb2/usbUhci.h`, `usb2/usbUhcdScheduleQueue.h`, `usb2/usbUhcdScheduleQSupport.h`, `usb2/usbUhcdCommon.h`, `usb2/usbUhcdSupport.h`

**usbUhcdScheduleQWaitForSignal**

**NAME**

`usbUhcdScheduleQWaitForSignal` – USB UHCD HCD ISR support routines

**ROUTINES**

**DESCRIPTION**

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

**INCLUDE FILES**

`usb2/usbOsal.h`, `usb2/usbHst.h`, `usb2/usbUhcdCommon.h`, `usb2/usbUhcdScheduleQueue.h`, `usb2/usbUhcdScheduleQSupport.h`, `usb2/usbUhci.h`
usbUhcdScheduleQueue

NAME
usbUhcdScheduleQueue – USB UHCD HCD schedule 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
usb2/usbOsal.h, usb2/usbHst.h, usb2/usbUhci.h, usb2/usbUhcdScheduleQueue.h,
usb2/usbUhcdSupport.h, usb2/usbUhcdCommon.h, usb2/usbUhcdScheduleQueue.h

usbd

NAME
usbd – USBD Routines

ROUTINES
usbdInit() – initializes USBD2.0
usbdExit() – exits USBD2.0
usbHstDriverRegister() – register class driver
usbHstDriverDeregister() – deregisters USB class driver
usbHstHCDRegister() – register Host Controller Driver
usbHstHCDDeregister() – deregister a Host Controller Driver
usbHstBusRegister() – registers an USB Bus
usbHstBusDeregister() – deregister a USB Bus
1 Libraries

usbdMisc

DESCRIPTION
This file initializes the various global variables for USBD module to come up and provides registration interfaces to the class driver and Host Controller Driver. This file includes the urb.c and device.c source files.

INCLUDE FILES
usb2/usbd.h

usbdMisc

NAME
usbdMisc – user specific information

ROUTINES
usbdBusCntGet() – obtain the number of USB’s attached to the host
usbdRootNodeIDGet() – obtain node id of the root hub on a Host Controller
usbdFrameWindowGet() – obtain Frame Window and current frame number from HCD

DESCRIPTION
This File contains the various interfaces required for obtaining user specific information from USBD.

INCLUDE FILES
usb2/usbdMisc.h

usrUsbAudioDemo

NAME
usrUsbAudioDemo – USB Audio Demo

ROUTINES
spkrAttachCallback() – receives attach callbacks from speaker SEQ_DEV driver
audioThread() – Dumps audio data to usbSpeakerLib
enterThreadSpkr() – waits for user to press [enter]
waitForSpeaker() – waits for a speaker to be connected
play() – sends a .wav file to a speaker
stop() – halts the playing of a wav
disVol() – displays volume levels in hex
up() – increases the volume
down() – decreases the volume
usbSpkrInit() – initializes USB speaker SEQ_DEV driver
usrUsbAudioDemo() – Entry point to USB audio demo
usbSpkrExit() – de-initializes USB speaker SEQ_DEV driver

DESCRIPTION
This file demonstrates the use of the USB stack, either the UHCI or OHCI host controller driver, and the USB audio class driver. It is assumed that this source will be included.
using the project facility to pull in the `INCLUDE_USB_AUDIO_DEMO`. In addition this file assumes the USB stack has already been initialized and a host controller has been attached. Finally, this demonstration also makes the assumption that the target system has wav files stored on a ATA type device.

USB audio Demo instructions

When your vxWorks image boots, use the demo by entering any of the following commands at the vxWorks prompt:

- `play [file]`  
  Plays a wave file. The file name is a file that exists in the current working directory.
- `stop`  
  Stops song that is currently playing
- `up`  
  Raises the volume level
- `down`  
  Lowers the volume level
- `mute`  
  Brings the volume level to a tolerable level
- `disVol`  
  Shows the current volume level

This demo works with any of the supported USB speakers listed in the USB Developer's Kit User's Guide

INCLUDE FILES

none

**usrUsbBulkDevInit**

**NAME**

`usrUsbBulkDevInit` – USB Mass Storage Bulk driver initialization

**ROUTINES**

- `usrUsbRawRead()` – raw read from the device
- `usrUsbRawWrite()` – raw write to the device
- `usrUsbBulkShow()` – shows routine for displaying all bulk devices.
- `usrUsbBulkDriveEmpty()` – routine to check if drive has media inserted.
- `usrUsbBulkDevDown()` – un-initializes USB BULK Mass storage driver
- `usbBulkXbdTest()` – Test to see if media is present or not. If a change
- `usbBulkXbdEject()` – Eject the device and instantiate the next step

**DESCRIPTION**

This configlette initializes the USB Mass Storage (Bulk-Only) class driver and places it in the driver table. On boot, it can be referred to by the name given specified by the
usrUsbCbiUfiDevInit

NAME
usrUsbCbiUfiDevInit – USB Mass Storage CBI driver initialization

ROUTINES
usrUsbCbiUfiDevInit() – initializes USB UFI Mass storage driver.
usrUsbCbiUfiDevDown() – un-initializes USB UFI Mass storage driver.
usbCbiXbdMediaTest() – Test for media existence
usbCbiXbdTest() – Test to see if media is present or not. If a change
usbCbiXbdEject() – Eject the device and instantiate the next step

DESCRIPTION
This configlette initializes the USB Mass Storage Control / Bulk / Interrupt driver and places it in the driver table. On boot, it can be referred to by the name given specified by CBI_DRIVE_NAME. This assumes the USB host stack has already been initialized and has a host controller driver attached.

INCLUDE FILES
usb/usbdLib.h, usb/usbQueueLib.h, usbBulkDevLib.h, usbBulkDevLibP.h, fsMonitor.h

usrUsbHcdEhciInit

NAME
usrUsbHcdEhciInit – Initialization of a EHCI Host Controller Driver

ROUTINES
usrUsbHcdEhciAttach() – attaches EHCI HCD to the USB Stack
usrUsbHcdEhciDetach() – detaches the EHCI HCD from the USB Stack

DESCRIPTION
This configlette initializes a USB Host Controller if present on a system. The initialization process includes bringing up the chip and also “attaches” the chip to the previously initilazed USB stack. The stack is initialized in usrUsbInit.c

INCLUDE FILES
usb/usbPciLib.h, usb/pciConstants.h, usb/usbdLib.h, usb2/usbOsal.h, usb2/usbEhcdInitExit.h
usrUsbHcdOhciInit

NAME
usrUsbHcdOhciInit – Initialization of a OHCI Host Controller Driver

ROUTINES
usrUsbHcdOhciAttach() – attaches OHCI HCD to the USB Stack
usrUsbHcdOhciDetach() – detaches the OHCI HCD from the USB Stack

DESCRIPTION
This configlette initializes a USB Host Controller if present on a system. The initialization process includes bringing up the chip and also “attaches” the chip to the previously initialized USB stack. The stack is initialized in usrUsbInit.c

INCLUDE FILES
usb/usbPciLib.h, usb/pciConstants.h, usb/usbdLib.h, usb2/usbOsal.h, usb2/usbOhci.h

usrUsbHcdUhciInit

NAME
usrUsbHcdUhciInit – Initialization of a UHCI Host Controller Driver

ROUTINES
usrUsbHcdUhciAttach() – attaches UHCI HCD to the USB Stack
usrUsbHcdUhciDetach() – detaches UHCI HCD from the USB Stack

DESCRIPTION
This configlette initializes a USB UHCI Host Controller if present on a system. The initialization process includes bringing up the chip and also “attaches” the chip to the previously initialized USB stack. The stack is initialized in usrUsbInit.c.

INCLUDE FILES
usb/usbPciLib.h, usb/usbdLib.h, usb2/usbHst.h, usb2/usbUhci.h

usrUsbInit

NAME
usrUsbInit – Initialization of the USB Host stack

ROUTINES
usbdTranslationInit() – initialize the USBD 1.1 Translation Layer
usrUsbHubInit() – initialize the USB Hub Driver
usbInit() – initialize the USB2 stack
usbExit() – un - initialize the USB2 stack
usb2WindViewLog() – Log USB2 Host Stack WindView events
usbLogMsg() – Dumps the log messages
usrUsbKbdInit

NAME

usrUsbKbdInit – Initialization of the USB Keyboard driver

ROUTINES

usbKbdDevCreate() – create a VxWorks device for an USB keyboard
usbKbdDrvUnInit() – shuts down an I/O USB keyboard driver
playWithKeyboard() – prints key presses to the terminal

DESCRIPTION

This configlette initializes the USB keyboard driver. This assumes the USB host stack has already been initialized and has a host controller driver attached.

This configlette demonstrates how a user might integrate a USB class driver into the vxWorks file system. usbKbdDevCreate () installs a USB keyboard and its associated driver functions into the driver table allowing the keyboard to be accessed with standard fopen, close, read, write, etc. type calls.

There is an included test routine playWithKeyboard () that demonstrates using the ios functionality.

INCLUDE FILES

iosLib.h, vxWorks.h, iv.h, ioLib.h, tyLib.h, intLib.h,errnoLib.h, sioLib.h, stdlib.h, stdio.h, logLib.h, selectLib.h, drv/usb/usbKeyboardLib.h, usb/usbHid.h

usrUsbMseInit

NAME

usrUsbMseInit – Initialization of the USB Mouse driver

ROUTINES

usbMseDrvUnInit() – shuts down an I/O USB mouse driver
usrUsbMseInit() – initialize the USB mouse driver

DESCRIPTION

This configlette initializes the USB mouse driver. This assumes the USB host stack has already been initialized and has a host controller driver attached.

This configlette demonstrates how a user might integrate a USB class driver into the vxWorks file system. usbMseDevCreate () installs a USB mouse and its associated driver
functions into the driver table allowing the mouse to be accessed with standard fopen, close, read, write, etc. type calls.

**INCLUDE FILES**  
errnoLib.h, drv/usb/usbMouseLib.h

---

**usrUsbPciInit**

**NAME**  
usrUsbPciInit – USB Host Controller host bus specific initialization

**ROUTINES**

**DESCRIPTION**

This configlette initializes a USB Host Controller (OHCI, UHCI, EHCI) host bus interface for both PCusb2EhciPciInit - to initialize EHCI host bus specific initializationI and non-PCI based Host Controllers.

In the USB 2.0 host stack, the Host Controller Drivers assume that the host controllers follow Little Endian format. This is true for PCI based host controllers and non-PCI host controllers in LE format. However, it does not hold good for non-PCI host controllers in Big Endian architecture.

A structure *usbHcdBusInfo* has been introduced to contain function pointers used by host controllers to perform bus specific operations and the existing BADDR and IRQ values.

Here is a more detailed explanation of the function pointers:

- **pFuncDataSwap** is the function pointer used for register read/write conversions and the data structure conversions. The function *usbDataSwap()* can be registered to perform data swap for the PCI based host controllers and the non-PCI host controllers which follow LE format. This function pointer should be NULL for non-PCI controllers following BE format.

- **pFuncCpuToBus** is used for converting the CPU address to the bus specific address.

- **pFuncBusToCpu** is used for converting the bus specific address to CPU address.

- **pFuncIntConnect** connects an ISR to the interrupt line. *usbPciIntConnect* should be registered as the function to connect an ISR in the case of PCI based host controllers. For non-PCI based host controllers, a user-defined function should be registered by *usbPciStub.c.sysEhciPciInit* - to initialize EHCI host PCI bus before MMU initialization

- **pFuncIntDisconnect** disconnects an ISR from the interrupt line. *usbPciIntRestore()* should be registered as the function to disconnect an ISR in the case of PCI based host controllers. For non-PCI based host controllers, a user-defined function should be registered by *usbPciStub.c.*

- **pFuncBufferSwap** is used for buffer contents swapping. For some non-PCI host controllers which follow BE format, *usbBufferSwap()* can be registered as the function pointer for
pFuncBufferSwap. However, depend on hardware, it may not be necessary. For example, the on-board OHCI host controller on ms7727se need to have buffer swapped. However, the on-board OHCI host controller on pb1000 does not need to have buffer swapped. **usrUsbPciInit()** should be NULL for PCI based host controllers and non-PCI host controllers following LE format.

**pDeviceSpecificData** can be used to store device specific data.

**NON-PCI and PCI BASED HOST CONTROLLER INITIALIZATION**

The information given below is only for one type of host controller — OHCI. This can be extended to other types of host controllers.

Arrays of pointers to **usbHcdBusInfo** data structure are defined for all host controller types. Each array element corresponds to a single host controller.

Since non-PCI host controller is initializes before PCI based host controller. The initialization of non-PCI host controller is discussed first.

**NON-PCI BASED HOST CONTROLLER INITIALIZATION**

BSP specific function **usbPciClassFind()** is called to retrieve information about the non-PCI host controllers on the target board. This has to be called before initializing the PCI.

The prototype of the **usbPciClassFind()** function is as follows:

```c
BOOL usbPciClassFind
(     
UINT8 pciClass,             /* PCI device class */
UINT8 subClass,             /* PCI device sub-class */
UINT8 pgmIf,                /* Programming interface */
UINT16 index,               /* Caller wants nth matching dev */
pUINT8 pBusNo,              /* Bus number of matching dev */
pUINT8 pDeviceNo,           /* Device number of matching dev */
pUINT8 pFuncNo              /* Function number of matching dev */
)
```

In the case of PCI based OHCI host controller, the value of 0x10 will be passed as the 3rd parameter. In the case of non-PCI OHCI host controllers, bit 7 of the 3rd parameter should be set to 1. For PCI based host controllers, bit 7 should be set to 0.

For non-PCI controller, the 1st, 2nd and 4th parameters will be the same as that of the PCI host controllers. It is for maintaining backward compatibility. The 5th parameter, **pBusNo** will be the pointer to the **usbHcdBusInfo** data structure This function populates all the members of the data structure.

If no non-PCI host controller is present in the target, this function returns FALSE, otherwise TRUE.

**PCI BASED HOST CONTROLLER INITIALIZATION**

The PCI host controllers reside after the non-PCI host controllers in the arrays maintained. The index into the array of **usbHcdBusInfo** is updated based on the number of non-PCI host
controllers. All the members of the `usbHcdBusInfo` should be initialized for PCI based host controllers.

**BSP UPDATES**

The existing BSPs which support PCI based controllers does not need to be modified.

The existing BSPs which support non-PCI based controllers (using USB 1.1.x stack) may need change in the `target/config/<BSP>/usbPciStub.c` file.

The function `usbPciClassFind()` has to support non-PCI controllers also. It should check the value of bit 7 in `pgmIf`. If it is set to 1, then update the members of `usbHcdBusInfo` data structure which is passed as “pBusNo” and return TRUE. If bit 7 of `pciClass` is set to 0, then call `pciFindClass()` for PCI initialization.

New function `usbNonPciCpuToBus()` needs to be defined which converts the CPU address to the bus specific address. The prototype of this function is:

```c
UINT32 usbNonPciCpuToBus
{
    pVOID pMem             /* CPU address to convert */
}
```

However, `usbNonPciCpuToBus` definition is only necessary if the BSP supports both PCI and on-PCI USB host controllers. If the BSP supports only non-PCI host controllers, the existing interface for PCI which performs the conversion from CPU address to PCI address (`usbMemToPci()`) should be used.

New function `usbNonPciBusToMem()` needs to be defined which converts the bus specific address to the CPU address. The prototype of this function is:

```c
pVOID usbNonPciBusToMem
{
    UINT32 pciAdrs            /* bus address to be converted */
}
```

However, `usbNonPciBusToMem` definition is only necessary if the BSP support both PCI and non-PCI USB host controllers. If the BSP supports only non-PCI host controllers, the existing interface for PCI which performs the conversion from PCI address to CPU address (`usbPciToMem()`) should be used.

New function `usbNonPciIntConnect()` has to be defined which disconnects an ISR connected to an interrupt line. The prototype of this function is:
VOID usbNonPciIntRestore
(
    INT_HANDLER_PROTOTYPE func,    /@ int handler to be removed @/
    pVOID param,                    /@ parameter for int handler @/
    UINT16 intNo                    /@ interrupt vector number @/
)

However, **usbNonPciIntRestore** definition is only necessary if the BSP support both PCI and non-PCI USB host controllers. If the BSP supports only non-PCI host controllers, the existing interface for PCI based host controllers (**usbPciIntRestore**) should be used.

**NAME**

usrUsbPegasusEndInit

**ROUTINES**

sysUsbEndPegasusLoad() – load (create) USB END device
usbLoadPegasus() – load (create) USB END device
usbPegasusEndStart() – make the network device operational
usbPegasusAttachCallback() – configuration level callback
pegasusClientThread() – handles device insertions / removals.
usrUsbPegasusEndInit() – initialize the USB END Pegasus driver
usrUsbPegasusEndDown() – un-initialize the USB END Pegasus driver

**DESCRIPTION**

This configlette initializes the USB END pegasus driver. This assumes the USB host stack has already been initialized and has a host controller driver attached.

**INCLUDE FILES**
ipProto.h, usb/ossLib.h, usb/usbQueueLib.h, drv/usb/usbPegasusEnd.h

**NAME**

usrUsbPnInit

**ROUTINES**

usbPnDrvUnInit() – shuts down an I/O USB printer driver
usrUsbPnInit() – initialize the USB printer driver
usbPnWrTest() – test the USB printer write function

**DESCRIPTION**

This configlette initializes the USB Printer driver.
**DESCRIPTION**
This configlette initializes the USB printer driver. This assumes the USB host stack has already been initialized and has a host controller driver attached.

This configlette demonstrates how a user might integrate a USB class driver into the vxWorks file system. `usbPrtmDevCreate()` installs a USB printer and its associated driver functions into the driver table allowing the printer to be accessed with standard fopen, close, read, write, etc. type calls. The decision to keep this functionality out of the driver itself was made for backwards compatibility reasons.

**INCLUDE FILES**
- iosLib.h
- vxWorks.h
- iv.h
- ioLib.h
- tyLib.h
- intLib.h
- erntoLib.h
- sioLib.h
- stdlib.h
- stdio.h
- logLib.h
- selectLib.h
- drv/usb/usbPrinterLib.h
- usb/usbPrinter.h

---

**usrUsbSpkrInit**

**NAME**
`usrUsbSpkrInit` – Initialization of the USB Speaker driver

**ROUTINES**
- `usbSpkrDevCreate()` – create a VxWorks device for an USB speaker
- `usbSpkrDrvUnInit()` – shuts down an I/O USB speaker driver
- `usrUsbSpkrInit()` – initialize the USB speaker driver

**DESCRIPTION**
This configlette initializes the USB speaker driver. This assumes the USB host stack has already been initialized and has a host controller driver attached.

This configlette demonstrates how a user might integrate a USB class driver into the vxWorks file system. `usbSpkrDevCreate()` installs a USB speaker and its associated driver functions into the driver table allowing the speaker to be accessed with standard fopen, close, read, write, etc. type calls. The decision to keep this functionality out of the driver itself was made for backwards compatibility reasons.

**INCLUDE FILES**
- iosLib.h
- vxWorks.h
- iv.h
- ioLib.h
- tyLib.h
- intLib.h
- erntoLib.h
- sioLib.h
- stdlib.h
- stdio.h
- logLib.h
- selectLib.h
- drv/usb/usbSpeakerLib.h
- usb/usbAudio.h
- usb/tools/wavFormat.h

---

**usrUsbTargInit**

**NAME**
`usrUsbTargInit` – WindView Instrumentation function Definition

**ROUTINES**
- `usbPeriphWindViewLog()` – USB2.0 Peripheral Stack WindView Log Events
- `usbPeriLogMsg()` – dumps the log messages

**DESCRIPTION**
This file consist of the function definition for USB 2.0 Peripheral Stack WindView Events.
usrUsbTargKbdInit

NAME
usrUsbTargKbdInit – Initialization of the keyboard function driver

ROUTINES
usrUsbTargKbdInit() – initialization function for keyboard.

DESCRIPTION
This file contains the initialization function for the keyboard function driver on the peripheral.

INCLUDE FILES
string.h, wvLib.h, usb/usbPlatform.h, usb/target/usbHalCommon.h, usb/target/usbTcdPdiusb12EvalLib.h, usb/target/usbTargLib.h, usb/usbHid.h, drv/usbd12EvalLib.h, usb/target/usbTargKbdLib.h

usrUsbTargMsInit

NAME
usrUsbTargMsInit – Initialization of the keyboard function driver

ROUTINES
usbTargMsInit() – initializes USB mass storage functionality driver

DESCRIPTION
This file contains the initialization function for the Mass Storage function driver on the peripheral.

INCLUDE FILES
vxWorks.h, stdio.h, blkIo.h, usb/usbPlatform.h, usb/usb.h, usb/usbLib.h, usb/target/usbTargLib.h, drv/usbd12EvalLib.h, drv/usbLib.h, drv/usbBulkDevLib.h, drv/usbd12EvalLib.h, drv/target/usbIspl582Eval.h, drv/target/usbIspl582EvalLib.h, drv/target/usbIspl582Tcd.h, drv/target/usbIspl582Debug.h, drv/target/usbTcdNET2280Lib.h

usrUsbTargPciInit

NAME
usrUsbTargPciInit – Initialization of USB Target Controller PCI Interfaces

INCLUDE FILES
string.h, wvLib.h
ROUTINES

sysIspl582PciInit() – to configure the PCI interface for ISP 1582
sysNET2280PciInit() – to map the PCI base addresses to memory
sys2NET2280PciInit() – to configure the PCI interface for NET2280

DESCRIPTION

This configlette initializes a Philips ISP 1582 & NetChip NET2280 Peripheral Controller PCI Interface.

INCLUDE

usb/pciConstants.h, usb/usbPcilib.h

---

**usrUsbTargPrnInit**

NAME

usrUsbTargPrnInit – Initialization of the printer function driver

ROUTINES

usrUsbTargPrnInit() – Initialization function for printer.

DESCRIPTION

This file contains the initialization function for the printer function driver on the peripheral.

INCLUDE FILES

usb/usbPlatform.h, usb/target/usbHalCommon.h,
drv/usb/target/usbTcdPduusbldEvalLib.h, usb/target/usbTargLib.h,
drv/usb/target/usbTargPrnLib.h

---

**usrUsbTool**

NAME

usrUsbTool – USB Exerciser.

ROUTINES

usbTool() – Primary entry point for USB bus exerciser.

DESCRIPTION

usbTool is a command line-driven program which allows the user to exercise USB capabilities for the USB Host and the Peripheral Stack.

In the vxWorks environment, the user starts usbTool by invoking the “usbTool” entry point from the vxWorks shell. usbTool then displays a prompt:

```
USB>
```

The user may now enter commands which will be parsed and executed by usbTool. Each command follows the format:

```
USB>command [optional parameter(s)]
```
Commands are executed after the user presses [enter]. The user may enter the "help" or "?" commands to see a list of currently supported commands.

Multiple commands may be entered on the same command line separated by semicolons (;). usbTool which execute 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).

The "quit"/"exit"/"bye" command terminates usbTool.

**Include Files**

- stdio.h
- stdLib.h
- string.h
- ioLib.h
- ctype.h
- usb/usbPlatform.h
- usb/tools/cmdParser.h
- usb/usbPciLib.h
- usb/usbdlLib.h
- usb2/usbHst.h
- usb2/usbHubInitialization.h
- usrUsbInit.c
- drv/usbUhci.h
- drv/usbOhci.h
- usb2/usbEhcdInitExit.h
- usb/usbHid.h
- drv/usb/usbKeyboardLib.h
- drv/usb/usbMouseLib.h
- usb/usbPrinter.h
- drv/usb/usbPrinterLib.h
- usb/tools/wavFormat.h
- usb2/usbAudio.h
- drv/usb/usbSpeakerLib.h
- usb/target/usbHalCommon.h
- usb/target/usbTargLib.h
- drv/usb/target/usbTargKbdLib.h
- drv/usb/target/usbTargPmnLib.h
- drv/usb/target/usbPdiusbd12Eval.h
- drv/usb/target/usbTcdPdiusbd12EvalLib.h
- drv/usb/target/usbTargPhilipsD12EvalLib.h
- drv/usb/target/usbTcdIsp1582EvalLib.h
- drv/usb/target/usbTcdNET2280Lib.h
Routines

BufferPageCross() – checks if there is page cross in transfer buffer .................. 75
CmdParserExitFunc() – Terminates parser execution........................................ 75
CmdParserHelpFunc() – Displays list of supported commands.......................... 76
ExecCmd() – Execute the command line .................................................... 76
GetHexToken() – Retrieves value of hex token............................................. 77
GetNextToken() – Retrieves the next token from an input string .................... 78
KeywordMatch() – Compare keywords................................................................ 78
PromptAndExecCmd() – Prompt for a command and execute it ....................... 79
SkipSpace() – Skips leading white space in a string .................................. 79
TruncSpace() – Truncates string to eliminate trailing whitespace ................ 80
audioThread() – Dumps audio data to usbSpeakerLib .................................. 80
bulkInErpCallbackCSW() – send the CSW on bulk-in pipe .......................... 81
bulkInErpCallbackData() – process end of data phase on bulk-in pipe ........... 81
bulkOutErpCallbackCBW() – process the CBW on bulk-out pipe .................. 82
bulkOutErpCallbackData() – process end of data phase on bulk-out pipe ....... 82
disVol() – displays volume levels in hex.................................................... 83
down() – decreases the volume.................................................................... 83
enterThreadSpkr() – waits for user to press [enter] ........................................ 83
ossCalloc() – Allocates memory initialized to zeros .................................... 84
ossFree() – Master USB memory free routine ............................................. 84
ossInitialize() – Initializes ossLib .............................................................. 85
ossMalloc() – Master USB memory allocation routine .................................. 85
ossMemUsedGet() – Retrieves amount of memory currently in use by USB .... 86
ossMutexCreate() – Creates a new mutex...................................................... 86
ossMutexDestroy() – Destroys a mutex........................................................ 86
ossMutexRelease() – Releases (gives) a mutex............................................. 87
ossMutexTake() – Attempts to take a mutex.................................................. 87
ossOldFree() – Frees globally allocated memory ........................................ 88
ossOldInstall() – Installs old method of USB malloc and free ....................... 88
ossOldMalloc() – Global memory allocation.................................................. 89
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ossPartFree()</td>
<td>Frees globally allocated memory</td>
<td>89</td>
</tr>
<tr>
<td>ossPartIdGet()</td>
<td>Retrieves the partition ID of USB memory partition</td>
<td>90</td>
</tr>
<tr>
<td>ossPartMalloc()</td>
<td>USB memory allocation</td>
<td>90</td>
</tr>
<tr>
<td>ossPartSizeGet()</td>
<td>Retrieves the size of the USB memory partition</td>
<td>91</td>
</tr>
<tr>
<td>ossPartSizeSet()</td>
<td>Sets the the initial size of the USB memory partition</td>
<td>91</td>
</tr>
<tr>
<td>ossSemCreate()</td>
<td>Creates a new semaphore</td>
<td>92</td>
</tr>
<tr>
<td>ossSemDestroy()</td>
<td>Destroys a semaphore</td>
<td>92</td>
</tr>
<tr>
<td>ossSemGive()</td>
<td>Signals a semaphore</td>
<td>93</td>
</tr>
<tr>
<td>ossSemTake()</td>
<td>Attempts to take a semaphore</td>
<td>93</td>
</tr>
<tr>
<td>ossShutdown()</td>
<td>Shuts down ossLib</td>
<td>94</td>
</tr>
<tr>
<td>ossStatus()</td>
<td>Returns OK or ERROR and sets errno based on status</td>
<td>94</td>
</tr>
<tr>
<td>ossThreadCreate()</td>
<td>Spawns a new thread</td>
<td>94</td>
</tr>
<tr>
<td>ossThreadDestroy()</td>
<td>Attempts to destroy a thread</td>
<td>95</td>
</tr>
<tr>
<td>ossThreadSleep()</td>
<td>Voluntarily relinquishes the CPU</td>
<td>96</td>
</tr>
<tr>
<td>ossTime()</td>
<td>Returns relative system time in msec</td>
<td>96</td>
</tr>
<tr>
<td>pegasusClientThread()</td>
<td>handles device insertions / removals</td>
<td>97</td>
</tr>
<tr>
<td>pegasusMuxTxRestart()</td>
<td>place muxTxRestart on netJobRing</td>
<td>97</td>
</tr>
<tr>
<td>pegasusOutIrqInUse()</td>
<td>determines if any of the output IRP's are in use</td>
<td>97</td>
</tr>
<tr>
<td>play()</td>
<td>sends a .wav file to a speaker</td>
<td>98</td>
</tr>
<tr>
<td>playWithKeyboard()</td>
<td>prints key presses to the terminal</td>
<td>98</td>
</tr>
<tr>
<td>spkrAttachCallback()</td>
<td>receives attach callbacks from speaker SEQ_DEV driver</td>
<td>99</td>
</tr>
<tr>
<td>stop()</td>
<td>halts the playing of a wav</td>
<td>99</td>
</tr>
<tr>
<td>sys2NET2280PciInit()</td>
<td>to configure the PCI interface for NET2280</td>
<td>99</td>
</tr>
<tr>
<td>sysISP1582PciInit()</td>
<td>to configure the PCI interface for ISP 1582</td>
<td>100</td>
</tr>
<tr>
<td>sysNET2280PciInit()</td>
<td>to map the PCI base addresses to memory</td>
<td>100</td>
</tr>
<tr>
<td>sysUSBEndPegasusLoad()</td>
<td>load (create) USB END device</td>
<td>101</td>
</tr>
<tr>
<td>up()</td>
<td>increases the volume</td>
<td>101</td>
</tr>
<tr>
<td>usb2WindViewLog()</td>
<td>Log USB2 Host Stack WindView events</td>
<td>102</td>
</tr>
<tr>
<td>usbBulkBlkDevCreate()</td>
<td>create a block device</td>
<td>102</td>
</tr>
<tr>
<td>usbBulkDevInit()</td>
<td>registers USB Bulk only mass storage class driver</td>
<td>103</td>
</tr>
<tr>
<td>usbBulkDevIoctl()</td>
<td>perform a device-specific control</td>
<td>104</td>
</tr>
<tr>
<td>usbBulkDevLock()</td>
<td>Marks USB_BULK_DEV structure as in use</td>
<td>104</td>
</tr>
<tr>
<td>usbBulkDevShow()</td>
<td>shows routine for displaying all LUNs of a device</td>
<td>105</td>
</tr>
<tr>
<td>usbBulkDevShutDown()</td>
<td>shuts down the USB bulk-only class driver</td>
<td>105</td>
</tr>
<tr>
<td>usbBulkDevUnlock()</td>
<td>Marks USB_BULK_DEV structure as unused</td>
<td>106</td>
</tr>
<tr>
<td>usbBulkDevEmpty()</td>
<td>routine to check if drive has media inserted</td>
<td>106</td>
</tr>
<tr>
<td>usbBulkDevShow()</td>
<td>shows routine for displaying one LUN of a device</td>
<td>107</td>
</tr>
<tr>
<td>usbBulkDynamicAttachRegister()</td>
<td>Register SCSI/BULK-ONLY device attach callback</td>
<td>107</td>
</tr>
<tr>
<td>usbBulkDynamicAttachUnregister()</td>
<td>Unregisters SCSI/BULK-ONLY attach callback</td>
<td>107</td>
</tr>
<tr>
<td>usbBulkGetMaxLun()</td>
<td>Return the max LUN number for a device</td>
<td>108</td>
</tr>
<tr>
<td>usbBulkShow()</td>
<td>shows routine for displaying all bulk devices</td>
<td>108</td>
</tr>
<tr>
<td>usbBulkXbdEject()</td>
<td>Eject the device and instantiate the next step</td>
<td>109</td>
</tr>
<tr>
<td>usbBulkXbdTest()</td>
<td>Test to see if media is present or not. If a change</td>
<td>110</td>
</tr>
<tr>
<td>usbCbiUfiBlkDevCreate()</td>
<td>create a block device</td>
<td>110</td>
</tr>
<tr>
<td>usbCbiUfiDevInit()</td>
<td>registers USB CBI mass storage class driver for UFI devices</td>
<td>111</td>
</tr>
</tbody>
</table>
Routines

- `usbCbiUfiDevIoctl()` – perform a device-specific control
- `usbCbiUfiDevLock()` – Marks `CBI_UFI_DEV` structure as in use
- `usbCbiUfiDevShutDown()` – shuts down the USB CBI mass storage class driver
- `usbCbiUfiDevUnlock()` – Marks `CBI_UFI_DEV` structure as unused
- `usbCbiUfiDynamicAttachRegister()` – Register UFI device attach callback
- `usbCbiUfiDynamicAttachUnregister()` – Unregisters CBI_UFI attach callback
- `usbCbiXbdEject()` – Eject the device and instantiate the next step
- `usbCbiXbdMediaTest()` – Test for media existence
- `usbConfigCountGet()` – Retrieves number of device configurations
- `usbConfigDescrGet()` – reads full configuration descriptor from device
- `usbDescrCopy()` – copies descriptor to a buffer
- `usbDescrCopy32()` – copies descriptor to a buffer
- `usbDescrParse()` – search a buffer for the a particular USB descriptor
- `usbDescrParseSkip()` – search for a descriptor and increment buffer
- `usbDescrStrCopy()` – copies an ASCII string to a string descriptor
- `usbDescrStrCopy32()` – copies an ASCII string to a string descriptor
- `usbEhcdExit()` – uninitializes the EHCI Host Controller
- `usbEhcdInit()` – initializes the EHCI Host Controller
- `usbEhcdRHCancelURB()` – cancels a request submitted for an endpoint
- `usbEhcdRHDeletePipe()` – deletes a pipe specific to an endpoint
- `usbEhcdRHSubmitURB()` – submits a request to an endpoint
- `usbEhcdRhClearPortFeature()` – clears a feature of the port
- `usbEhcdRhCreatePipe()` – creates a pipe specific to an endpoint
- `usbEhcdRhGetHubDescriptor()` – get the hub descriptor
- `usbEhcdRhGetPortStatus()` – get the status of the port
- `usbEhcdRhProcessClassSpecificRequest()` – processes a class specific request
- `usbEhcdRhProcessControlRequest()` – processes a control transfer request
- `usbEhcdRhProcessInterruptRequest()` – processes an interrupt transfer request
- `usbEhcdRhProcessStandardRequest()` – processes a standard transfer request
- `usbEhcdRhSetPortFeature()` – set the features of the port
- `usbExit()` – un-initialize the USB2 stack
- `usbHalTcdAddressSet()` – hal interface to set address
- `usbHalTcdAttach()` – attaches a TCD
- `usbHalTcdCurrentFrameGet()` – hal interface to get Current Frame Number
- `usbHalTcdDetach()` – detaches a TCD
- `usbHalTcdDeviceFeatureClear()` – hal interface to clear feature on device
- `usbHalTcdDeviceFeatureSet()` – hal interface to set feature on the device
- `usbHalTcdDisable()` – disables the target controller
- `usbHalTcdEnable()` – enables the target controller
- `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
- `usbHalTcdErpCancel()` – cancel an ERP
usbHalTcdErpSubmit() – submit an ERP for an endpoint ................................. 135
usbHalTcdSignalResume() – hal interface to initiate resume signal. .............. 136
usbHandleCreate() – Creates a new handle .............................................. 136
usbHandleDestroy() – Destroys a handle .................................................. 137
usbHandleInitialize() – Initializes the handle utility library .................... 137
usbHandleShutdown() – Shuts down the handle utility library .................... 138
usbHandleValidate() – Validates a handle .................................................. 138
usbHidIdleSet() – Issues a SET_IDLE request to a USB HID ...................... 139
usbHidProtocolSet() – Issues a SET_PROTOCOL request to a USB HID .......... 139
usbHidReportSet() – Issues a SET_REPORT request to a USB HID ............. 140
usbHstBusDeregister() – deregister a USB Bus ......................................... 140
usbHstBusRegister() – registers an USB Bus ............................................ 141
usbHstClearFeature() – clear feature USB request .................................... 141
usbHstDriverDeregister() – deregisters USB class driver ........................... 142
usbHstDriverRegister() – register class driver .......................................... 142
usbHstGetConfiguration() – get configuration USB request ....................... 143
usbHstGetDescriptor() – get USB Descriptor ............................................. 143
usbHstGetFrameNumber() – get USB frame number .................................... 144
usbHstGetInterface() – get interface USB request ...................................... 144
usbHstGetStatus() – Get USB Status ......................................................... 145
usbHstHCDDeregister() – deregister a Host Controller Driver .................... 145
usbHstHCDRegister() – register Host Controller Driver ............................. 146
usbHstResetDevice() – called by the class drivers to reset its device ............. 146
usbHstSelectiveResume() – resume device .................................................. 147
usbHstSelectiveSuspend() – suspend device ............................................... 147
usbHstSetBitRate() – set USB bit rate ....................................................... 148
usbHstSetConfiguration() – set configuration USB request ....................... 148
usbHstSetDescriptor() – set descriptor USB request ................................... 149
usbHstSetFeature() – set feature USB request .......................................... 149
usbHstSetInterface() – set interface USB request ...................................... 150
usbHstSetSynchFrame() – set synch frame USB request .............................. 150
usbHstURBCancel() – cancel USB Request Block ....................................... 151
usbHstURBSubmit() – submit USB request block ...................................... 151
usbHubExit() – de-registers and cleans up the USB Hub Class Driver ............ 152
usbHubInit() – registers USB Hub Class Driver function pointers .................. 152
usbInit() – initialize the USB2 stack ......................................................... 152
usbKbdDevCreate() – create a VxWorks device for an USB keyboard ............. 153
usbKbdDrvUnInit() – shuts down an I/O USB keyboard driver ...................... 154
usbKeyboardDevInit() – initialize USB keyboard SIO driver ....................... 154
usbKeyboardDevShutdown() – shuts down keyboard SIO driver .................... 155
usbKeyboardDynamicAttachRegister() – Register keyboard attach callback ...... 155
usbKeyboardDynamicAttachUnregister() – Unregisters keyboard attach callback 156
usbKeyboardSioChanLock() – Marks SIO_CHAN structure as in use ............... 156
usbKeyboardSioChanUnlock() – Marks SIO_CHAN structure as unused .......... 157
usbListFirst() – Returns first entry on a linked list .................................... 158
usbListLink() – Add an element to a linked list ................................................... 158
usbListLinkProt() – Add an element to a list guarded by a mutex .................. 159
usbListNext() – Retrieves next pStruct in a linked list ................................... 159
usbListUnlink() – Remove an entry from a linked list ................................... 160
usbListUnlinkProt() – Removes an element from a list guarded by a mutex ...... 160
usbLoadPegasus() – load (create) USB END device .......................................... 161
usbLogMsg() – Dumps the log messages .............................................................. 161
usbMemToPci() – Convert a memory address to a PCI-reachable memory address 162
usbMouseDevInit() – initialize USB mouse SIO driver ...................................... 162
usbMouseDevShutdown() – shuts down mouse SIO driver .................................. 163
usbMouseDynamicAttachRegister() – Register mouse attach callback .......... 163
usbMouseDynamicAttachUnregister() – Unregisters mouse attach callback ... 164
usbMouseSioChanLock() – Marks SIO_CHAN structure as in use ...................... 164
usbMouseSioChanUnlock() – Marks SIO_CHAN structure as unused ................. 165
usbMsBulkInErpInUseFlagGet() – get the Bulk-in ERP inuse flag .................... 166
usbMsBulkInErpInUseFlagSet() – set the Bulk-In ERP inuse flag ..................... 166
usbMsBulkInErpInit() – initialize the bulk-in ERP ........................................... 167
usbMsBulkInStall() – stall the bulk-in pipe ...................................................... 167
usbMsBulkInUnStall() – unstall the bulk-in pipe .............................................. 168
usbMsBulkOutErpInUseFlagGet() – get the Bulk-Out ERP inuse flag ............... 168
usbMsBulkOutErpInUseFlagSet() – set the Bulk-Out ERP inuse flag ................ 169
usbMsBulkOutErpInit() – initialize the bulk-Out ERP .................................... 169
usbMsBulkOutStall() – stall the bulk-out pipe ................................................ 170
usbMsBulkOutUnStall() – unstall the bulk-out pipe ........................................ 170
usbMsCBWGet() – get the last mass storage CBW received ............................. 170
usbMsCSWGet() – get the current CSW ............................................................ 171
usbMsCSWinInit() – initialize the CSW ............................................................ 171
usbMsIsConfigured() – test if the device is configured .................................. 172
usbMsTestRxCallback() – invoked after test data is received ......................... 172
usbMsTestTxCallback() – invoked after test data transmitted ......................... 173
usbMsEDrvUnInit() – shuts down an I/O USB mouse driver ......................... 173
usbOhciDumpEndpointDescriptor() – dump endpoint descriptor contents ........ 173
usbOhciDumpGeneralTransferDescriptor() – dump general transfer descriptor .... 174
usbOhciDumpMemory() – dump memory contents ............................................. 174
usbOhciDumpPendingTransfers() – dump pending transfers .......................... 175
usbOhciDumpPeriodicEndpointList() – dump periodic endpoint descriptor list .... 175
usbOhciDumpRegisters() – dump registers contents ....................................... 176
usbOhciExit() – uninitialise the USB OHCI Host Controller Driver .................. 176
usbOhciInit() – initialise the USB OHCI Host Controller Driver .................... 177
usbOhciInitializeModuleTestingFunctions() – obtains entry points .................. 177
usbPciByteGet() – Returns a UINT8 configuration value .................................. 178
usbPciByteIn() – input a byte from PCI I/O space .......................................... 178
usbPciByteOut() – output a byte to PCI I/O space .......................................... 179
usbPciClassFind() – Locates PCI devices by class ........................................... 179
usbPciConfigHeaderGet() – Reads a device’s PCI configuration header .............................. 180
usbPciDwordGet() – Returns a UINT32 configuration value........................................ 180
usbPciDwordIn() – input a dword from PCI I/O space .............................................. 181
usbPciDwordOut() – outputs a dword to PCI I/O space ........................................... 181
usbPciIntConnect() – Connect to a interrupt vector ............................................... 182
usbPciIntDisconnect() – Removes an interrupt handler .......................................... 182
usbPciMemFlush() – Flush a region of memory through the cache ........................... 183
usbPciMemInvalidate() – Invalidate cache for a region of memory ............................ 183
usbPciMemIoOffset() – Return PCI MEMIO to CPU MEMIO offset .............................. 184
usbPciToMem() – Convert a PCI-reachable address to a CPU-reachable pointer ............ 184
usbPciWordGet() – Returns a UINT16 configuration value..................................... 185
usbPciWordIn() – input a word from PCI I/O space ................................................ 185
usbPciWordOut() – outputs a word to PCI I/O space .............................................. 186
usbPegasusAttachCallback() – configuration level callback .................................. 186
usbPegasusDevLock() – marks USB_PEGASUS_DEV structure as in use .................... 187
usbPegasusDevUnlock() – marks USB_PEGASUS_DEV structure as unused .............. 187
usbPegasusDynamicAttachRegister() – register PEGASUS device attach callback ...... 188
usbPegasusDynamicAttachUnregister() – unregisters PEGASUS attach callbacks ......... 189
usbPegasusEndInit() – initializes the pegasus library ............................................. 189
usbPegasusEndLoad() – initialize the driver and device ........................................ 190
usbPegasusEndStart() – make the network device operational ................................ 191
usbPegasusEndUnInit() – un-initializes the pegasus class driver .............................. 191
usbPegasusReadReg() – read contents of specified and print .................................. 192
usbPeriLogMsg() – dumps the log messages ............................................................ 192
usbPeriphWindViewLog() – USB2.0 Peripheral Stack WindView Log Events ............... 193
usbPrinterDevInit() – initialize USB printer SIO driver ......................................... 193
usbPrinterDevShutdown() – shuts down printer SIO driver ..................................... 194
usbPrinterDynamicAttachRegister() – Register printer attach callback ................. 194
usbPrinterDynamicAttachUnregister() – Unregisters printer attach callback ............. 195
usbPrinterSioChanLock() – Marks SIO_CHAN structure as in use ............................ 195
usbPrinterSioChanUnlock() – Marks SIO_CHAN structure as unused ....................... 196
usbPrnDrvUnInit() – shuts down an I/O USB printer driver .................................... 196
usbQueueCreate() – Creates a O/S-independent queue structure .............................. 197
usbQueueDestroy() – Destroys a queue ................................................................. 198
usbQueueGet() – Retrieves a message from a queue .............................................. 198
usbQueuePut() – Puts a message onto a queue ..................................................... 199
usbRecurringTime() – calculates recurring time for interrupt/isoch transfers .......... 200
usbSpeakerDevInit() – initialize USB speaker SIO driver ....................................... 200
usbSpeakerDevShutdown() – shuts down speaker SIO driver .................................. 201
usbSpeakerDynamicAttachRegister() – Register speaker attach callback ................ 201
usbSpeakerDynamicAttachUnregister() – Unregisters speaker attach callback .......... 202
usbSpeakerSeqDevLock() – Marks SEQ_DEV structure as in use .............................. 203
usbSpeakerSeqDevUnlock() – Marks SEQ_DEV structure as unused ....................... 203
usbSpkrDevCreate() – create a VxWorks device for an USB speaker ...................... 204
usbSpkrDrvUnInit () – shuts down an I/O USB speaker driver..................................... 204
usbSpkrExit () – de-initializes USB speaker SEQ_DEV driver.............................. 205
usbSpkrInit () – initializes USB speaker SEQ_DEV driver................................... 205
usbTargControlPayloadRcv () – receives data on the default control pipe.......... 206
usbTargControlResponseSend () – sends data to host on the control pipe........ 206
usbTargControlStatusSend () – sends control transfer status to the host.......... 207
usbTargCurrentFrameGet () – retrieves the current USB frame number.................. 207
usbTargDeviceFeatureClear () – clears a specific feature...................................... 208
usbTargDeviceFeatureSet () – sets or enable a specific feature.......................... 208
usbTargDisable () – disables a target channel.................................................. 209
usbTargEnable () – enables target channel onto USB......................................... 209
usbTargInitialize () – initializes the USB Target Library.................................. 210
usbTargKbdCallbackInfo () – returns usbTargKbdLib callback table.................... 210
usbTargKbdInjectReport () – injects a 'boot report'......................................... 211
usbTargMgmtCallback () – invoked when HAL detects a management event........ 212
usbTargMsCallbackInfo () – returns usbTargPrnLib callback table..................... 212
usbTargMsInit () – initializes USB mass storage functionality driver.................. 213
usbTargPipeCreate () – creates a pipe for communication on an endpoint.......... 213
usbTargPipeDestroy () – destroys an endpoint pipe......................................... 214
usbTargPipeStatusGet () – returns the endpoint status.................................... 214
usbTargPipeStatusSet () – sets pipe stalled/unstalled status............................. 215
usbTargPrnCallbackInfo () – returns buffer status/info...................................... 215
usbTargPrnDataInfo () – restarts listening ERP.............................................. 216
usbTargPrnDataRestart () – create an RBC BLK_DEV device................................ 217
usbTargRbcBlockDevCreate () – return opaque pointer to the RBC BLK I/O DEV device .... 217
usbTargRbcBlockDevGet () – set the pointer to the RBC BLK I/O DEV device structure.... 218
usbTargRbcBufferWrite () – write micro-code to the RBC device.......................... 218
usbTargRbcCacheSync () – read the capacity of the RBC device............................ 219
usbTargRbcCapacityRead () – format the RBC device......................................... 220
usbTargRbcFormat () – retrieve inquiry data from the RBC device...................... 220
usbTargRbcInquiry () – select the mode parameter page of the RBC device.......... 221
usbTargRbcModeSelect () – retrieve sense data from the RBC device.................... 221
usbTargRbcModeSense () – send reserve data to the host................................... 222
usbTargRbcPersistentReserveIn () – reserve resources on the RBC device........... 222
usbTargRbcPersistentReserveOut () – prevent or allow the removal of the RBC device ...... 223
usbTargRbcRead () – release a resource on the RBC device............................... 224
usbTargRbcRelease () – request sense data from the RBC device....................... 224
usbTargRbcRequestSense () – reserve a resource on the RBC device.................... 225
usbTargRbcReserve () – start or stop the RBC device........................................ 225
usbTargRbcStartStop () – test if the RBC device is ready.................................... 226
usbTargRbcTestUnitReady () – vendor specific call........................................... 226
usbTargRbcVendorSpecific () – verify the last data written to the RBC device........ 227
usbTargRbcVerify ()
usbTargRbcWrite() – write to the RBC device
usbTargSetupErpCallback() – handles the setup packet
usbTargShutdown() – shutdown the USB target library
usbTargSignalResume() – drives RESUME signalling on USB
usbTargTcdAttach() – to attach the TCD to the target library
usbTargTcdDetach() – detaches a USB target controller driver
usbTargTransfer() – to transfer data through a pipe
usbTargTransferAbort() – cancels a previously submitted USB_ERP
usbTcdIsp1582EvalExec() – single Entry Point for ISP 1582 TCD
usbTcdNET2280Exec() – single Entry Point for NETCHIP 2280 TCD
usbTcdPdiusbd12EvalExec() – single entry point for PDIUSBD12 TCD
usbTool() – Primary entry point for USB bus exerciser
usbTransferTime() – Calculates bus time required for a USB transfer
usbUhcdExit() – uninitialise the USB UHCD Host Controller Driver
usbUhcdInit() – initialise the USB UHCD Host Controller Driver
usbAddressGet() – gets the USB address for a given device
usbAddressSet() – sets the USB address for a given device
usbBusCntGet() – obtain the number of USB's attached to the host
usbBusCountGet() – get number of USBs attached to the host
usbBusStateSet() – Sets bus state (e.g., suspend/resume)
usbClientRegister() – Registers a new client with the USBD
usbClientUnregister() – Unregisters a USBD client
usbConfigurationGet() – gets USB configuration for a device
usbConfigurationSet() – sets USB configuration for a device
usbCurrentFrameGet() – returns the current frame number for a USB
usbDescriptorGet() – retrieves a USB descriptor
usbDescriptorSet() – sets a USB descriptor
usbDynamicAttachRegister() – Registers client for dynamic attach notification
usbDynamicAttachUnRegister() – Unregisters client for attach notification
usbExit() – exits USBD2.0
usbFeatureClear() – clears a USB feature
usbFeatureSet() – sets a USB feature
usbFrameWindowGet() – obtain Frame Window and current frame number from HCD
usbHcdAttach() – attaches an HCD to the USBD
usbHcdDetach() – Detaches an HCD from the USBD
usbHubPortCountGet() – returns number of ports connected to a hub
usbInit() – initializes USBD2.0
usbInitialize() – Initialize the USBD
usbInterfaceGet() – retrieves a device's current interface
usbInterfaceSet() – sets a device's current interface
usbMgmtCallbackSet() – sets management callback for a client
usbNodeIdGet() – gets the id of the node connected to a hub port
usbNodeInfoGet() – returns information about a USB node
usbPipeCreate() – Creates a USB pipe for subsequent transfers
usbPipeDestroy() – Destroys a USB data transfer pipe
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>usbdRootNodeIDGet</code></td>
<td>obtain node id of the root hub on a Host Controller</td>
<td>257</td>
</tr>
<tr>
<td><code>usbdRootNodeIDGet</code></td>
<td>returns root node for a specific USB</td>
<td>258</td>
</tr>
<tr>
<td><code>usbdShutdown</code></td>
<td>Shuts down the USBD</td>
<td>258</td>
</tr>
<tr>
<td><code>usbdStatisticsGet</code></td>
<td>Retrieves USBD operating statistics</td>
<td>259</td>
</tr>
<tr>
<td><code>usbdStatusGet</code></td>
<td>Retrieves USB status from a device/interface/etc</td>
<td>259</td>
</tr>
<tr>
<td><code>usbdSynchFrameGet</code></td>
<td>Returns a device's isochronous synch. frame</td>
<td>260</td>
</tr>
<tr>
<td><code>usbdTransfer</code></td>
<td>Initiates a transfer on a USB pipe</td>
<td>261</td>
</tr>
<tr>
<td><code>usbdTransferAbort</code></td>
<td>Aborts a transfer</td>
<td>263</td>
</tr>
<tr>
<td><code>usbdTranslationInit</code></td>
<td>initialize the USBD 1.1 Translation Layer</td>
<td>263</td>
</tr>
<tr>
<td><code>usbdVendorSpecific</code></td>
<td>Allows clients to issue vendor-specific USB requests</td>
<td>264</td>
</tr>
<tr>
<td><code>usbdVersionGet</code></td>
<td>Returns USBD version information</td>
<td>265</td>
</tr>
<tr>
<td><code>usbtuDataUrbCompleteCallback</code></td>
<td>Callback called on URB completion</td>
<td>265</td>
</tr>
<tr>
<td><code>usbtuDataVendorSpecificCallback</code></td>
<td>Callback called on Vendor Specific Request</td>
<td>266</td>
</tr>
<tr>
<td><code>usbtuInitClientIrpCompleteThreadFn</code></td>
<td>Client Thread Function</td>
<td>266</td>
</tr>
<tr>
<td><code>usbtuInitClientThreadFn</code></td>
<td>Client Thread Function</td>
<td>267</td>
</tr>
<tr>
<td><code>usbtuInitDeviceAdd()</code></td>
<td>Device Attach Callback</td>
<td>267</td>
</tr>
<tr>
<td><code>usbtuInitDeviceRemove()</code></td>
<td>Device Detach Callback</td>
<td>268</td>
</tr>
<tr>
<td><code>usbtuInitDeviceResume()</code></td>
<td>Device Resume Callback</td>
<td>268</td>
</tr>
<tr>
<td><code>usbtuInitDeviceSuspend()</code></td>
<td>Device Suspend Callback</td>
<td>268</td>
</tr>
<tr>
<td><code>usbtuInitThreadFn</code></td>
<td>Translation Unit Thread Function</td>
<td>269</td>
</tr>
<tr>
<td><code>usrUsbAudioDemo()</code></td>
<td>Entry point to USB audio demo</td>
<td>270</td>
</tr>
<tr>
<td><code>usrUsbBulkDevDown()</code></td>
<td>un-initializes USB BULK Mass storage driver</td>
<td>270</td>
</tr>
<tr>
<td><code>usrUsbBulkDriveEmtpy()</code></td>
<td>routine to check if drive has media inserted</td>
<td>270</td>
</tr>
<tr>
<td><code>usrUsbBulkShow()</code></td>
<td>shows routine for displaying all bulk devices</td>
<td>271</td>
</tr>
<tr>
<td><code>usrUsbCbiUfiDevDown()</code></td>
<td>un-initializes USB UFI Mass storage driver</td>
<td>271</td>
</tr>
<tr>
<td><code>usrUsbCbiUfiDevInit()</code></td>
<td>initializes USB UFI Mass storage driver</td>
<td>272</td>
</tr>
<tr>
<td><code>usrUsbHcdEhciAttach()</code></td>
<td>attaches EHCI HCD to the USB Stack</td>
<td>272</td>
</tr>
<tr>
<td><code>usrUsbHcdEhciDetach()</code></td>
<td>detaches the EHCI HCD from the USB Stack</td>
<td>272</td>
</tr>
<tr>
<td><code>usrUsbHcdOhciAttach()</code></td>
<td>attaches OHCI HCD to the USB Stack</td>
<td>272</td>
</tr>
<tr>
<td><code>usrUsbHcdOhciDetach()</code></td>
<td>detaches the OHCI HCD from the USB Stack</td>
<td>273</td>
</tr>
<tr>
<td><code>usrUsbHcdUhciAttach()</code></td>
<td>attaches UHCI HCD to the USB Stack</td>
<td>273</td>
</tr>
<tr>
<td><code>usrUsbHcdUhciDetach()</code></td>
<td>detaches UHCI HCD from the USB Stack</td>
<td>274</td>
</tr>
<tr>
<td><code>usrUsbHubInit()</code></td>
<td>initialize the USB Hub Driver</td>
<td>274</td>
</tr>
<tr>
<td><code>usrUsbMseInit()</code></td>
<td>initialize the USB mouse driver</td>
<td>275</td>
</tr>
<tr>
<td><code>usrUsbPegasusEndDown()</code></td>
<td>un-initialize the USB END Pegasus driver</td>
<td>275</td>
</tr>
<tr>
<td><code>usrUsbPegasusEndInit()</code></td>
<td>initialize the USB END Pegasus driver</td>
<td>276</td>
</tr>
<tr>
<td><code>usrUsbPrntInit()</code></td>
<td>initialize the USB printer driver</td>
<td>276</td>
</tr>
<tr>
<td><code>usrUsbRawRead()</code></td>
<td>raw read from the device</td>
<td>276</td>
</tr>
<tr>
<td><code>usrUsbRawWrite()</code></td>
<td>raw write to the device</td>
<td>277</td>
</tr>
<tr>
<td><code>usrUsbSpkrInit()</code></td>
<td>initialize the USB speaker driver</td>
<td>277</td>
</tr>
<tr>
<td><code>usrUsbTargKbdInit()</code></td>
<td>initialization function for keyboard</td>
<td>278</td>
</tr>
<tr>
<td><code>usrUsbTargPrtiInit()</code></td>
<td>Initialization function for printer</td>
<td>278</td>
</tr>
<tr>
<td><code>waitForSpeaker()</code></td>
<td>waits for a speaker to be connected</td>
<td>279</td>
</tr>
</tbody>
</table>
BufferPageCross()

NAME BufferPageCross() – checks if there is page cross in transfer buffer

SYNOPSIS

BOOL BufferPageCross

    (UINT32 uHostControllerIndex,
     PUSB_OHCI_ISOCHRONOUS_TRANSFER_DESCRIPTOR pTransferDescriptor,
     UINT32 uPacketBufferOffset)

DESCRIPTION This function checks the start and end of the transfer buffer, see if they
is page cross in transfer buffer.

PARAMETERS

uHostControllerIndex (IN) - index of the host controller.
pTransferDescriptor (IN) - Pointer to transfer descriptor.
uPacketBufferOffset (IN) - Packet buffer offset.

RETURNS TRUE if there is page cross, FALSE if not.

ERRNO None.

SEE ALSO usbOhciTransferManagement

CmdParserExitFunc()

NAME CmdParserExitFunc() – Terminates parser execution

SYNOPSIS

UINT16 CmdParserExitFunc

    (pVOID param,    /* Generic parameter passed */
     char **ppCmd,  /* Ptr to remainder of cmd line */
     FILE *fin,     /* stream for input (if any) */
     FILE *fout)    /* stream for output (if any) */

DESCRIPTION Returns RET_OK, causing the parser to return RET_OK to the caller signally normal
termination of the parser.

RETURNS RET_OK
CmdParserHelpFunc()

NAME
CmdParserHelpFunc() – Displays list of supported commands

SYNOPSIS
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) */
}

DESCRIPTION
Displays the list of commands in the parser command table to fout. When the parser recognizes that this function is about to be executed, it substitutes a pointer to the current CMD_DESCR table in param. If this function is called directly, param 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 */
}

ERRNO
None.

SEE ALSO
cmdParser
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 pCmd line, interprets it as a hex value, and stores the result in pToken. If there are no remaining tokens, stores defVal in pToken instead.

RETURNS
Pointer into pStr following end of copied pToken

ERRNO
None.

SEE ALSO
cmdParser
GetNextToken()

NAME
GetNextToken() – Retrieves the next token from an input string

SYNOPSIS
cchar *GetNextToken
{
    char *pStr, /* Input string */
    char *pToken, /* Bfr to receive token */
    UINT16 tokenLen /* Max length of Token bfr */
}

DESCRIPTION
Copies the next token from pStr to pToken. White space before the next token is discarded. Tokens are delimited by white space and by the command separator, CMD_SEPARATOR. No more than tokenLen - 1 characters from pStr will be copied into pToken. tokenLen must be at least one and pToken will be NULL terminated upon return.

RETURNS
Pointer into pStr following end of copied pToken.

ERRNO
None.

SEE ALSO
cmdParser

KeywordMatch()

NAME
KeywordMatch() – Compare keywords

SYNOPSIS
int KeywordMatch
{
    char *s1, /* string 1 */
    char *s2, /* string 2 */
    int len /* max length to compare */
}

DESCRIPTION
Compares s1 and s2 up to len 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 s1 and s2 are the same
-n if s1 < s2
+n if s1 > s2

ERRNO
None.

SEE ALSO
cmdParser
PromptAndExecCmd( )

NAME
PromptAndExecCmd( ) – Prompt for a command and execute it.

SYNOPSIS
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 */)

DESCRIPTION
Displays pPrompt to fout and prompts for input from fin. Then, parses/executes the command. 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

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
Truncates 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

audioThread()

NAME
audioThread() – Dumps audio data to usbSpeakerLib

DESCRIPTION
By convention, param is the file handle for the file to be played and the global "wavDataLength" should be the length of the data chunk. The file position should be set to the beginning of the data in the data chunk.
This thread closes the file after reading all data.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo
bulkInErpCallbackCSW()

NAME    bulkInErpCallbackCSW() – send the CSW on bulk-in pipe

SYNOPSIS void bulkInErpCallbackCSW
               (pVOID erp /* USB_ERP endpoint request packet */)

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 void bulkInErpCallbackData
               (pVOID erp /* USB_ERP endpoint request packet */)

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  void bulkOutErpCallbackCBW
           (  
               pVOID erp /* USB_ERP endpoint request packet */  
           )

DESCRIPTION  This routine processes 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  void bulkOutErpCallbackData
           (  
               pVOID erp /* USB_ERP endpoint request packet */  
           )

DESCRIPTION  This routine is invoked following a data OUT phase from the host.

RETURNS  N/A

ERRNO  none

SEE ALSO  usbTargRbcLib
disVol()

NAME
disVol() – displays volume levels in hex

SYNOPSIS
STATUS disVol (void)

DESCRIPTION
none

RETURNS
OK or ERROR

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo

down()

NAME
down() – decreases the volume

SYNOPSIS
STATUS down (void)

DESCRIPTION
none

RETURNS
OK or ERROR

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo

enterThreadSpkr()

NAME
enterThreadSpkr() – waits for user to press [enter]

DESCRIPTION
none

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo
ossCalloc()  

NAME  
ossCalloc() – Allocates memory initialized to zeros

SYNOPSIS  
pVOID ossCalloc
    (  
        UINT32 numBytes /* size of buffer to allocate */  
    )

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  
void ossFree
    (  
        pVOID bfr  
    )

DESCRIPTION  
ossFree() calls the free routine installed in the global variable ossFreeFuncPtr. This defaults to ossPartFree(), but can be changed by the user 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  
STATUS ossInitialize (void)

DESCRIPTION  
This function should be called once at initialization in order to initialize the ossLib. Calls to this function may be nested. This permits multiple, independent libraries to use this library without need to coordinate 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  
void * ossMalloc
{
    UINT32 numBytes
}

DESCRIPTION  
ossMalloc() calls the malloc routine installed in the global variable ossMallocFuncPtr. These default to ossPartMalloc(), but can be changed by the user 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 amount of memory currently in use by USB.

SYNOPSIS
UINT32 ossMemUsedGet (void)

DESCRIPTION
Returns the amount, in bytes, currently being used by USB.

RETURNS
Number of bytes of memory in use.

ERRNO
None.

SEE ALSO
ossLib

ossMutexCreate()

NAME
ossMutexCreate() – Creates a new mutex

SYNOPSIS
STATUS ossMutexCreate
    (pMUTEX_HANDLE pMutexHandle /* Handle of newly created mutex */)

DESCRIPTION
This function creates a new mutex and returns the handle of that mutex in pMutexHandle. 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
ossMutexRelease()

NAME
ossMutexRelease() – Releases (gives) a mutex

SYNOPSIS
STATUS ossMutexRelease
    ( MUTEK_HANDLE mutexHandle /* Mutex to be released */ )

DESCRIPTION
Release 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
    ( MUTEK_HANDLE mutexHandle, /* Mutex to take */
    UINT32 blockFlag /* specifies blocking action */ )

DESCRIPTION
Destroys the mutex mutexHandle created by ossMutexCreate().

RETURNS
OK or ERROR

ERRNO
S_ossLib_GENERAL_FAULT

SEE ALSO
ossLib
DESCRIPTION
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. blockFlag 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 blockFlag are interpreted as a count of milliseconds to wait for the mutex to be released before declaring an error.

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 void ossOldFree
             ( void * bfr )

DESCRIPTION ossOldFree() frees memory allocated by ossMalloc().

RETURNS N/A

ERRNO None.

SEE ALSO ossLib

ossOldInstall()

NAME ossOldInstall() – Installs old method of USB malloc and free.

SYNOPSIS void ossOldInstall (void)
**DESCRIPTION**
Installs old method of USB malloc and free. This must be called before the call to `usbdInitialize()`.

**RETURNS**
N/A

**ERRNO**
None.

**SEE ALSO**
ossLib

---

### ossOldMalloc()

**NAME**
ossOldMalloc() – Global memory allocation

**SYNOPSIS**
```c
void * ossOldMalloc
    (UINT32 numBytes /* Size of buffer to allocate */)
```

**DESCRIPTION**
`ossOldMalloc()` allocates a buffer of `numBytes` 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 or 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**
```c
void ossPartFree
    (pVOID bfr)
```

**DESCRIPTION**
`ossPartFree()` frees memory allocated by `ossMalloc()`.
ossPartIdGet( )

NAME  ossPartIdGet( ) – Retrieves the partition ID of USB memory partition.

SYNOPSIS  PART_ID ossPartIdGet (void)

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  void * ossPartMalloc
            (UINT32 numBytes /* Size of buffer to allocate */)

DESCRIPTION  ossPartMalloc( ) allocates cache-safe buffer of size numBytes 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 is defaulted to 64k but can be modified to suit the users needs. This partition will dynamically grow based on additional need. Memory allocated by this function must be freed by calling ossFree( ).

RETURNS  Pointer to allocated buffer, or NULL

ERRNO  None.

SEE ALSO  ossLib
### ossPartSizeGet()

**NAME**

ossPartSizeGet() – Retrieves the size of the USB memory partition.

**SYNOPSIS**

```c
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 initial size of the USB memory partition.

**SYNOPSIS**

```c
STATUS ossPartSizeSet
    (   
        UINT32 numBytes
    )
```

**DESCRIPTION**

Sets the size of the USB memory partition. This must be called prior to the first call to ossMalloc. This will set the size that ossMalloc will use to do 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 via 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
2 Routines
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 function should be called once at system shutdown if an 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 status & 0xffff is not equal to zero, then sets errno to the indicated status and returns ERROR. Otherwise, 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

94
The `ossThreadCreate()` function 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**

```c
STATUS ossThreadDestroy
(    THREAD_HANDLE threadHandle  /* handle of thread to be destroyed */
);
```

**DESCRIPTION**

This function attempts to destroy the thread specified by `threadHandle`.

**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 STATUS ossThreadSleep

{       UINT32 msec /* Number of msec to sleep */
}

DESCRIPTION Threads may call ossThreadSleep() to voluntarily release the CPU to another
thread/process. If the msec argument is 0, then the thread will be reschuled for execution as
soon as possible. If the msec 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 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 approximately every 49 days, so time calculations should always be
based on the difference between two time values.

RETURNS relative system time in msec.

ERRNO None.

SEE ALSO ossLib
pegasusClientThread()

NAME  pegasusClientThread() – handles device insertions / removals.
SYNOPSIS  void pegasusClientThread(void)
DESCRIPTION  This routine is the configuration level handler that monitors device insertions and removals. Upon device connection, it starts the USB network device.
RETURNS  N/A
ERRNO  none
SEE ALSO  usrUsbPegasusEndInit

pegasusMuxTxRestart()

NAME  pegasusMuxTxRestart() – place muxTxRestart on netJobRing
SYNOPSIS  void pegasusMuxTxRestart
            (  
               END_OBJ * pEndObj  /* pointer to DRV_CTRL structure */  
            )
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  BOOL pegasusOutIrpInUse
(PEGASUS_DEVICE * pDevCtrl)

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

---

**play()**

**NAME**
play() – sends a .wav file to a speaker

**SYNOPSIS**
STATUS play
{
    char * fileName
}

**DESCRIPTION**
none

**RETURNS**
OK or ERROR

**ERRNO**
Not Available

**SEE ALSO**
usrUsbAudioDemo

---

**playWithKeyboard()**

**NAME**
playWithKeyboard() – prints key presses to the terminal

**SYNOPSIS**
STATUS playWithKeyboard (void)

**DESCRIPTION**
This is a test-application which prints the key pressed to the terminal.

**RETURNS**
OK or ERROR
**spkrAttachCallback()**

**NAME**
spkrAttachCallback() – receives attach callbacks from speaker SEQ_DEV driver

**DESCRIPTION**
none

**RETURNS**
N/A

**ERRNO**
Not Available

**SEE ALSO**
usrUsbAudioDemo

---

**stop()**

**NAME**
stop() – halts the playing of a wav

**SYNOPSIS**

```c
STATUS stop (void)
```

**DESCRIPTION**
none

**RETURNS**
OK or ERROR

**ERRNO**
Not Available

**SEE ALSO**
usrUsbAudioDemo

---

**sys2NET2280PciInit()**

**NAME**

sys2NET2280PciInit() – to configure the PCI interface for NET2280

**SYNOPSIS**

```c
void sys2NET2280PciInit (void)
```
### sysIsp1582PciInit()

**NAME**  
`sysIsp1582PciInit()` – to configure the PCI interface for ISP 1582

**SYNOPSIS**  
```c
void  sysIsp1582PciInit
  (long        * pBaseaddr,  /* base address */
   long        * pIrq        /* irq number */
)
```

**DESCRIPTION**  
This function is used to configure the PCI interface for ISP 1582. It obtains the PCI Configuration Header for the ISP 1582 and provides with the base address and the irq number.

**RETURNS**  
N/A

**ERRNO**  
none

**SEE ALSO**  
`usrUsbTargPciInit`

### sysNET2280PciInit()

**NAME**  
`sysNET2280PciInit()` – to map the PCI base addresses to memory

**SYNOPSIS**  
```c
void  sysNET2280PciInit (void)
```

This function is used to configure the PCI interface for NET2280. It obtains the PCI Configuration Header for the NET2280 and provides with the base addresses and the irq number.

3 PCI Base Addresses are obtained:

- PCI BASE ADDRESS [0] - for configuration registers
- PCI BASE ADDRESS [1] - for 8051 microcontroller
- PCI BASE ADDRESS [2] - for accessing the FIFO buffers directly

**RETURNS**  
N/A

**ERRNO**  
none

**SEE ALSO**  
`usrUsbTargPciInit`
sysUsbEndPegasusLoad()

NAME
sysUsbEndPegasusLoad() – load (create) USB END device

SYNOPSIS
END_OBJ * sysUsbPegasusEndLoad
(
    char * pParamStr, /* ptr to initialization parameter string */
    void * pDev        /* ptr to pegasus device */
)

DESCRIPTION
This routine loads the usb end device with initial parameters specified by values given in the BSP configuration files (config.h).

RETURNS
pointer to END object or ERROR.

ERRNO
none

SEE ALSO
usrUsbPegasusEndInit

up()

NAME
up() – increases the volume

SYNOPSIS
STATUS up (void)

DESCRIPTION
none

RETURNS
OK or ERROR

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo
usb2WindViewLog()

NAME usb2WindViewLog() – Log USB2 Host Stack WindView events

SYNOPSIS

```c
void usb2WindViewLog
    (UINT32 evId,
     char   *buffer,
     UINT32 mask)
```

DESCRIPTION

The USB2 Host Stack is instrumented to provide additional WindView User log events for
function entries/exits tracing within the Host Controllers. Logging is based whether or not
usbHcdWvFilter has been initialized with the appropriate filter mask. If WindView is not
enabled and the correct filter mask is set, then the WindView user event message will be
logged to the standard output.

```c
USB_EHCD_WV_FILTER  0x00000001
USB_OHCD_WV_FILTER   0x00000002
USB_UHCD_WV_FILTER   0x00000004
USB_USBD_WV_FILTER   0x00000008
USB_HUB_WV_FILTER    0x0000000F
```

RETURNS N/A

ERRNO none

SEE ALSO usrUsbInit

usbBulkBlkDevCreate()

NAME usbBulkBlkDevCreate() – create a block device

SYNOPSIS

```c
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**

```c
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
int usbBulkDevIoctl
    (XBD * pUsbBulkXbdDev, /* pointer to bulk device */
     int request,         /* request type */
     void * someArg       /* arguments related to request */
    )

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
STATUS usbBulkDevLock
    (USBD_NODE_ID nodeId  /* NodeId of the XBD to be marked as in use */
    )

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  void usbBulkDevShow
            (                
              USBD_NODE_ID nodeId /* nodeId of the bulk-only device */
            )

DESCRIPTION  This function displays all the logical unit number of the device specified by nodeId

RETURNS  N/A

ERRNO  none

SEE ALSO  usbBulkDevLib

usbBulkDevShutDown()

NAME  usbBulkDevShutDown() – shuts down the USB bulk-only class driver

SYNOPSIS  STATUS usbBulkDevShutDown
            (                
              int errCode /* Error code – reason for shutdown */
            )

DESCRIPTION  This routine unregisters the driver from USBD 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**
```c
STATUS usbBulkDevUnlock
    (USBD_NODE_ID nodeId  /* NodeId of the XBD to be marked as unused */)
```

**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**
```c
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 USBD client task’s context created for this driver.

RETURNS
OK, or ERROR if unable to register callback

ERRNO
S_usbBulkDevLib_BAD_PARAM
Bad Paramters 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 callback and arg 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
UINT8  usbBulkGetMaxLun
usbBulkShow()

NAME    usbBulkShow() – shows routine for displaying all bulk devices.

SYNOPSIS void usbBulkShow

DESCRIPTION This routine displays all the bulk devices connected

RETURNS N/A

ERRNO none

SEE ALSO usbBulkDevLib

usbBulkXbdEject()

NAME    usbBulkXbdEject() – Eject the device and instantiate the next step

SYNOPSIS int usbBulkXbdEject

DESCRIPTION /NOMANUAL

RETURNS Not Available

(int) nodeId /* nodeId of the bulk-only device */

DESCRIPTION This function returns the maximum LUN number of the device

RETURNS UINT8 value specifying the maximum LUN or 0, if nodeId not found

ERRNO none

SEE ALSO usbBulkDevLib
usbBulkXbdTest()

NAME

usbBulkXbdTest() – Test to see if media is present or not. If a change

SYNOPSIS

int usbBulkXbdTest
{
    pUSB_BULK_DEV_XBD_LUN pBulkDevLun
}

DESCRIPTION

of media has occurred, test XBD’s for 0 size and eject the device and instantiate the next
device accordingly.

/NOMANUAL

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

usrUsbBulkDevInit

usbCbiUfiBlkDevCreate()

NAME

usbCbiUfiBlkDevCreate() – create a block device

SYNOPSIS

XBD * usbCbiUfiBlkDevCreate
{
    USBD_NODE_ID nodeId /* Node Id of the CBI_UFI device */
}

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.
2 Routines

**usbCbiUfiDevInit()**

**NAME**

*usbCbiUfiDevInit()* – registers USB CBI mass storage class driver for UFI devices

**SYNOPSIS**

```c
STATUS usbCbiUfiDevInit (void)
```

**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 USBD.

**ERRNO**

- `S_usbCbiUfiDevLib_OUT_OF_RESOURCES`
  
  Resources are not available

- `S_usbCbiUfiDevLib_USBD_FAULT`
  
  USBD Fault has occured

**SEE ALSO**

*usbCbiUfiDevLib*

---

**usbCbiUfiDevIoctl()**

**NAME**

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

**SYNOPSIS**

```c
int usbCbiUfiDevIoctl( 
  XBD * pCbiUfiXbdDev, /* pointer to MSC/CBI/UFI device */
  int    request,  /* request type */
  void * someArg    /* arguments related to request */
)
```

**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
STATUS usbCbiUfiDevLock
    (USBD_NODE_ID nodeId  /* NodeId of the XBD to be marked as in use */ )

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**

```c
STATUS usbCbiUfiDevShutDown(  
    int errCode  /* Error code - reason for shutdown */  
)
```

**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**

```c
STATUS usbCbiUfiDevUnlock(  
    USBD_NODE_ID nodeId  /* NodeId of the XBD to be marked as unused */  
)
```

**DESCRIPTION**

This function releases a lock placed on a CBI_UFI_DEV structure. When a caller no longer needs a 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**

```c
STATUS usbCbiUfiDynamicAttachRegister
(
    USB_UFI_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:

```c
typedef (*USB_UFI_ATTACH_CALLBACK)
(  
    pVOID arg,
    USBD_NODE_ID cbiUfiDevId,
    UINT16 attachCode
);
```

`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 USBD client task's context created for this driver.

**RETURNS**

OK, or ERROR if unable to register callback

**ERRNO**

- `S_usbCbiUfiDevLib_BAD_PARAM` Bad Parameter passed
- `S_usbCbiUfiDevLib_OUT_OF_MEMORY` Sufficient memory not available

**SEE ALSO**

`usbCbiUfiDevLib`
2 Routines

usbCbiUfiDynamicAttachUnregister()

NAME

usbCbiUfiDynamicAttachUnregister() – Unregisters CBI_UFI attach callback.

SYNOPSIS

STATUS usbCbiUfiDynamicAttachUnregister
(  
  USB_UFI_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 CBI_UFI device attachment and removal. The callback and arg parameters 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

usbCbiXbdEject()

NAME

usbCbiXbdEject() – Eject the device and instantiate the next step

SYNOPSIS

int usbCbiXbdEject
(  
  pUSB_CBI_UFI_DEV pUsbCbiUfiDev
)

DESCRIPTION

/NOMANUAL

RETURNS

Not Available

ERRNO

Not Available

SEE ALSO

usrUsbCbiUfiDevInit
usbCbiXbdMediaTest()

NAME
usbCbiXbdMediaTest() – Test for media existence

SYNOPSIS
int usbCbiXbdMediaTest
   (   pUSB_CBI_UFI_DEV pUsbCbiUfiDev
   )

DESCRIPTION
/NOMANUAL

RETURNS
Not Available

ERRNO
Not Available

SEE ALSO
usrUsbCbiUfiDevInit

usbCbiXbdTest()

NAME
usbCbiXbdTest() – Test to see if media is present or not. If a change

SYNOPSIS
int usbCbiXbdTest
   (   pUSB_CBI_UFI_DEV pUsbCbiUfiDev
   )

DESCRIPTION
of media has occurred, test XBD’s for 0 size and eject the device and instantiate the next
device accordingly.
/NOMANUAL

RETURNS
Not Available

ERRNO
Not Available

SEE ALSO
usrUsbCbiUfiDevInit
usbConfigCountGet()  

NAME  
usbConfigCountGet() – Retrieves number of device configurations

SYNOPSIS  
STATUS usbConfigCountGet  
    (  
      USBD_CLIENT_HANDLE usbdClientHandle,  /* caller's USBD client handle */  
      USBD_NODE_ID nodeId,            /* device node ID */  
      pUINT16            pNumConfig         /* bfr to receive nbr of config */  
    )

DESCRIPTION  
Using the usbdClientHandle provided by the caller, this function reads the nodeId's device descriptor and returns the number of configurations supported by the device in pNumConfig.

RETURNS  
OK, or ERROR if unable to read device descriptor

ERRNO  
None.

SEE ALSO  
usbLib

usbConfigDescrGet()  

NAME  
usbConfigDescrGet() – reads full configuration descriptor from device

SYNOPSIS  
STATUS usbConfigDescrGet  
    (  
      USBD_CLIENT_HANDLE usbdClientHandle,  /* 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 */  
    )

DESCRIPTION  
This function reads the configuration descriptor cfgNo and all associated descriptors (interface, endpoint, etc.) for the device specified by nodeId. 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.

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 pBfr and pBfrLen. It is the caller’s responsibility to free the buffer using the OSS_FREE() macro.
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 */
)

RETURNS
N/A

ERRNO
None.

SEE ALSO
usbDescrCopyLib
**DESCRIPTION**

This function is the same as `usbDescrCopy()` except that `bfrLen` and `pActLen` refer to UINT32 quantities.

**RETTURNS**

N/A

**ERRNO**

None.

**SEE ALSO**

`usbDescrCopyLib`

---

### `usbDescrParse()`

**NAME**

`usbDescrParse()` – search a buffer for a particular USB descriptor

**SYNOPSIS**

```c
pVOID usbDescrParse
   (pUINT8 pBfr, /* buffer to parse */
    UINT16 bfrLen, /* length of buffer to parse */
    UINT8  descriptorType /* type of descriptor being sought */
   )
```

**DESCRIPTION**

Searches `pBfr` up to `bfrLen` bytes for a descriptor of type matching `descriptorType`. Returns a pointer to the descriptor if found.

**RETTURNS**

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**

```c
pVOID usbDescrParseSkip
   (pUINT8  *ppBfr, /* buffer to parse */
    pUINT16 pBfrLen, /* length of buffer to parse */
    UINT8   descriptorType /* type of descriptor being sought */
   )
```
DESCRIPTION
Searches ppBfr up to pBfrLen bytes for a descriptor of type matching descriptorType. Returns a pointer to the descriptor if found. ppBfr and pBfrLen 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
VOID usbDescrStrCopy
{
    pUINT8  pBfr,    /* destination buffer */
    char    *pStr,   /* source buffer */
    UINT16  bfrLen,  /* dest len */
    pUINT16 pActLen  /* actual length copied */
}

DESCRIPTION
This function constructs a properly formatted USB string descriptor in pBfr. The ASCII string pStr is copied to pBfr as a UNICODE string - as required by the USB spec. The actual length of the resulting descriptor is returned in pActLen if pActLen is non-NULL.

NOTE
The complete length of the string descriptor can be calculated as 2 * strlen (pStr) + 2. The pActLen will be the shorter of bfrLen or this value.

RETURNS
N/A

ERRNO
None.

SEE ALSO
usbDescrCopyLib
usbDescrStrCopy32()

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

SYNOPSIS VOID usbDescrStrCopy32

    (pUINT8  pBfr,    /* destination buffer */
     char    *pStr,   /* source buffer */
     UINT32  bfrLen,  /* dest len */
     pUINT32 pActLen  /* actual length copied */
    )

DESCRIPTION This function is the same as usbDescrStrCopy() except that bfrLen and pActLen refer to
UINT32 quantities.

RETURNS N/A

ERRNO None.

SEE ALSO usbDescrCopyLib

usbEhcdExit()

NAME    usbEhcdExit() – uninitializes the EHCI Host Controller

SYNOPSIS BOOLEAN usbEhcdExit(void)

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

SYNOPSIS
BOOLEAN usbEhcdInit(void)

DESCRIPTION
This routine initializes the EHCI Host Controller Driver and can be called from either the
target initialization code (bootup) or during runtime. The usbd and hub interfaces should
be initialized before this routine is called.

RETURNS
TRUE, or FALSE if there is an error during HCD initialization.

ERRNO
None.

SEE ALSO
usbEhcdInitExit

usbEhcdRHCcancelURB()

NAME
usbEhcdRHCcancelURB() – cancels a request submitted for an endpoint

SYNOPSIS
USBHST_STATUS usbEhcdRHCcancelURB
{
  pUSB_EHCD_DATA pHCDData, /* Ptr to HCD block */
  UINT32 uPipeHandle, /* Pipe Handle Identifier */
  pUSBHST_URB pURB /* Ptr to User Request Block */
}

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**

```c
USBHST_STATUS usbEhcdRHDeletePipe
    (pUSB_EHCD_DATA pHCDData,  /* Ptr to HCD block*/
    UINT32 uPipeHandle  /* Pipe Handle Identifier*/
    )
```

**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**

```c
USBHST_STATUS usbEhcdRHSubmitURB
    (pUSB_EHCD_DATA pHCDData,  /* Ptr to HCD block*/
    UINT32 uPipeHandle,  /* Pipe Handle Identifier*/
    pUSBHST_URB pURB,  /* Ptr to User Request Block*/
    )
```

**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**

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

**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**

```c
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 */
)
```

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

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 pUSBHST_URB  /* 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 request

SYNOPSIS

USBHST_STATUS usbEhcdRhProcessControlRequest

(pUSB_EHCD_DATA pHCDData,  /* Ptr to HCD block       */
pUSBHST_URB pUSBHST_URB  /* 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
2 Routines

** 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 **
USBHST_STATUS usbEhcdRhProcessStandardRequest

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

** DESCRIPTION **
This routine processes a standard 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
usbEhcdRhSetPortFeature()

NAME
usbEhcdRhSetPortFeature() – set the features of the port

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

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

usbExit()

NAME
usbExit() – uninitialized the USB2 Stack

SYNOPSIS
STATUS usbExit (void)

DESCRIPTION
This function un-intializes the USB2 Stack

RETURNS
OK if successful or ERROR if failure

ERRNO
none

SEE ALSO
usrUsbInit

usbHalTcdAddressSet()

NAME
usbHalTcdAddressSet() – hal interface to set address.

SYNOPSIS
STATUS usbHalTcdAddressSet
usbHalTcdAttach( )

NAME

usbHalTcdAttach() – attaches a TCD

SYNOPSIS

STATUS usbHalTcdAttach
{  USB_TCD_EXEC_FUNC tcdExecFunc,        /* single entry point of TCD */  pVOID tcdParam,           /* TCD specific paramter */  pUSBHAL_TCD_NEXUS pNexus,  /* TCD_NEXUS structure member */  pUSB_APPLN_DEVICE_INFO pDeviceInfo,  /* USB_APPLN_DEVICE_INFO */  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

STATUS usbHalTcdDeviceFeatureClear
(           /* USBHAL_TCD_NEXUS */
    pUSBHAL_TCD_NEXUS pNexus,  /* Feature to be cleared */
    UINT16 uFeatureSelector
)

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

STATUS usbHalTcdDeviceFeatureSet
(           /* USBHAL_TCD_NEXUS */
    pUSBHAL_TCD_NEXUS pNexus,  /* Feature to set */
    UINT16 uFeatureSelector,  /* Test Mode arguments */
    UINT8 uTestSelector
)

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**

```c
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 */
    )
```

**DESCRIPTION**

This function is used to configure an endpoint for USB operations. `pEndpointDesc` is the endpoint descriptor obtained from the above layer. On successful configuration, we get a pipe handle `ppPipeHandle` 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**

```c
STATUS usbHalTcdEndpointRelease
    (    pUSBHAL_TCD_NEXUS pNexus,      /* USBHAL_TCD_NEXUS */
        pVOID             pPipeHandle  /* pipe handle */
    )
```

**DESCRIPTION**

This function is used to release an endpoint configured earlier. `pPipeHandle` 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**

```c
STATUS usbHalTcdEndpointStateSet
    (pUSBHAL_TCD_NEXUS pNexus,       /* USBHAL_TCD_NEXUS */
     pVOID             pPipeHandle,  /* pipe handle */
     UINT16            state         /* state of the pipe */
    )
```

**DESCRIPTION**

This function is used to stall or un-stall an endpoint. *pPipeHandle* is the handle to the corresponding endpoint and *state* 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**

```c
STATUS usbHalTcdEndpointStatusGet
    (pUSBHAL_TCD_NEXUS pNexus,       /* USBHAL_TCD_NEXUS */
     pVOID             pPipeHandle,  /* pipe handle */
     pUINT8            pStatus       /* pointer to hold the endpoint status */
    )
```

**DESCRIPTION**

This function is used to get the status of an endpoint. *pPipeHandle* is the handle to the corresponding endpoint and *pStatus* 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**

STATUS usbHalTcdErpCancel

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

**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**

STATUS usbHalTcdErpSubmit

(  
  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
STATUS usbHalTcdSignalResume
    (    pUSBHAL_TCD_NEXUS pNexus /* USBHAL_TCD_NEXUS */ )

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
STATUS usbHandleCreate
    (    UINT32          handleSignature, /* Arbitrary handle signature */
        pVOID           handleParam, /* Arbitrary handle parameter */
        pGENERIC_HANDLE pHandle /* Newly allocated handle */ )

DESCRIPTION
Creates a new handle. The caller passes an arbitrary handleSignature and a handleParam.
The handleSignature 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**

```c
STATUS usbHandleDestroy
    (    GENERIC_HANDLE handle  /* handle to be destroyed */
    )
```

**DESCRIPTION**

This function destroys the handle 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**

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

**DESCRIPTION**

Initializes the handle utility library. Must be called at least once prior to 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.

`maxHandles` defines the maximum number of handles which should be allocated by the library. Passing a zero in `maxHandles` causes the library to allocate a default number of handles. `maxHandles` is ignored on nested calls to `usbHandleInitialize()`.

**RETURNS**

OK or ERROR

**ERRNO**

- `S_usbHandleLib_OUT_OF_MEMORY`
- `S_usbHandleLib_OUT_OF_RESOURCES`

**SEE ALSO**

`usbHandleLib`
usbHandleShutdown()

NAME
usbHandleShutdown() – Shuts down the handle utility library

SYNOPSIS
STATUS usbHandleShutdown (void)

DESCRIPTION
Shuts down the handle utility library. When calls to usbHandleInitialize() have been nested, usbHandleShutdown() must be called a corresponding number of times.

RETURNS
OK or ERROR

ERRNO
None.

SEE ALSO
usbHandleLib

usbHandleValidate()

NAME
usbHandleValidate() – Validates a handle

SYNOPSIS
STATUS usbHandleValidate
{
    GENERIC_HANDLE handle, /* handle to be validated */
    UINT32 handleSignature, /* signature used to validate handle */
    pVOID *pHandleParam /* Handle parameter on return */
}

DESCRIPTION
This function validates handle. The handle must match the handleSignature used when the handle was originally created. If the handle is valid, the pHandleParam 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
STATUS usbHidIdleSet
{
    USBD_CLIENT_HANDLE usbdClientHandle, /* caller's USBD client handle */
    USBD_NODE_ID nodeId,            /* desired node */
    UINT16 interface,             /* desired interface */
    UINT16 reportId,             /* desired report */
    UINT16 duration            /* idle duration */
}

DESCRIPTION
Using the usbdClientHandle provided by the caller, this function issues a SET_IDLE request to the indicated nodeId. The caller must also specify the interface, reportId, and duration. If the duration is zero, the idle period is infinite. If duration is non-zero, then it expresses time in 4msec units (e.g., a duration of 1 = 4msec, 2 = 8msec, 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
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 */
}

DESCRIPTION
Using the usbdClientHandle provided by the caller, this function issues a SET_PROTOCOL request to the indicated nodeId. The caller must specify the interface and the desired protocol. The protocol 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

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 */
)

DESCRIPTION

Using the usbdClientHandle provided by the caller, this function issues a SET_REPORT request to the indicated nodeId. The caller must also specify the interface, reportType, reportId, reportBfr, and reportLen. 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

usbd
**usbHstBusRegister()**

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

**SYNOPSIS**

```c
USBHST_STATUS usbHstBusRegister(
    UINT32 hHCDriver,    /* Host Controller Driver handle */
    UINT8  uSpeed,       /* USB Bus speed */
    UINT32 hDefaultPipe  /* Default pipe handle */
);
```

**DESCRIPTION**
This routine registers an USB Bus corresponding to the host controller.

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

**ERRNO**
None

**SEE ALSO**
usbd

**usbHstClearFeature()**

**NAME**
usbHstClearFeature() – clear feature USB request

**SYNOPSIS**

```c
USBHST_STATUS usbHstClearFeature(
    UINT32 hDevice,     /* USB device handle */
    UINT8  uRecipient,  /* Desired recipient */
    UINT16 uIndex,      /* Recipient index */
    UINT16 uFeature     /* Feature selector */
);
```

**DESCRIPTION**
This function is used to issue ClearFeature USB Standard Request

**RETURNS**
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, USBHST_INSUFFICIENT_MEMORY, USBHST_TIMEOUT if request submitted to Host controller Driver is timed out

**ERRNO**
None

**SEE ALSO**
urb
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 */
)

DESCRIPTION  
This routine registers the class driver with the USB Host Stack.

RETURNS  
USBHST_INVALID_PARAMETER, USBHST_SUCCESS, USBHST_FAILURE if Driver is already registered

ERRNO  
None

SEE ALSO  
usbd
usbHstGetConfiguration( )

NAME
usbHstGetConfiguration( ) – get configuration USB request

SYNOPSIS
USBHST_STATUS usbHstGetConfiguration
{
    UINT32 hDevice,  /* USB Device handle */
    UCHAR  *pBuffer  /* Ptr to fetched data buffer */
}

DESCRIPTION
This function is used to issue the GetConfiguration USB Standard Request

RETURNS
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, USBHST_INSUFFICIENT_MEMORY,
USBHST_TIMEOUT if request submitted to Host controller Driver is timed out

ERRNO
None

SEE ALSO
urb

usbHstGetDescriptor( )

NAME
usbHstGetDescriptor( ) – get USB Descriptor

SYNOPSIS
USBHST_STATUS usbHstGetDescriptor
{
    UINT32 hDevice,     /* USB device handle */
    UINT8  uDescType,   /* Type of descriptor */
    UINT8  uDescIndex,  /* Index of config or string descr */
    UINT16 swLangID,    /* Language ID in case of string descr */
    UINT32 *puSize,     /* Ptr to store number of bytes fetched */
    UCHAR  *pBuffer     /* Ptr to buffer to hold fetched data */
}

DESCRIPTION
This function is used to issue GET_DESCRIPTOR USB Standard Request.

RETURNS
USBHST_SUCCESS, USBHST_FAILURE if parameter is not valid

ERRNO
None

SEE ALSO
urb
usbHstGetFrameNumber( )

NAME
usbHstGetFrameNumber( ) – get USB frame number

SYNOPSIS
USBHST_STATUS usbHstGetFrameNumber
       (UINT32 hDevice,      /* Handle to requesting device */
       UINT16 * pFrameNumber  /* Ptr to hold frame number */
       )

DESCRIPTION
This function is called by the class driver layer to get the current frame number on bus

RETURNS
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, if Parameters are not valid

ERRNO
None

SEE ALSO
urb

usbHstGetInterface( )

NAME
usbHstGetInterface( ) – get interface USB request

SYNOPSIS
USBHST_STATUS usbHstGetInterface
       (UINT32 hDevice,           /* USB device handle          */
       UINT16 uInterfaceNumber,  /* Interface number           */
       UCHAR  *pBuffer           /* Ptr to fetched data buffer */
       )

DESCRIPTION
This function is used by the class driver layer to issue GetInterface USB Standard Request

RETURNS
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, USBHST_INSUFFICIENT_MEMORY,
USBHST_TIMEOUT if request submitted to Host controller Driver is timed out

ERRNO
None

SEE ALSO
urb
2 Routines

usbHstGetStatus() 

**NAME**

usbHstGetStatus() – Get USB Status

**SYNOPSIS**

```c
USBHST_STATUS usbHstGetStatus
{
    UINT32 hDevice, /* Handle to USB Device */
    UINT8 uRecipient, /* Desired recipient */
    UINT16 uIndex, /* Index of recipient */
    UCHAR *pBuffer /* Ptr to data buffer to hold data */
}
```

**DESCRIPTION**

This function is used to issue the GetStatus USB Standard request.

**RETURNS**

USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, USBHST_INSUFFICIENT_MEMORY, USBHST_TIMEOUT if request submitted to Host controller Driver is timed out.

**ERRNO**

None

**SEE ALSO**

urb

usbHstHCDDeregister() 

**NAME**

usbHstHCDDeregister() – deregister a Host Controller Driver

**SYNOPSIS**

```c
USBHST_STATUS usbHstHCDDeregister
{
    UINT32 hHCDriver /* Host Controller Driver handle */
}
```

**DESCRIPTION**

This routine deregisters a Host Controller Driver with the USB Stack.

**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

SYNOPSIS
USBHST_STATUS usbHstHCDRegister
    (pUSBHST_HC_DRIVER pHCDriver, /* Ptr to Host Controller driver */
     UINT32 *phHCDriver, /* Ptr to Host Controller handle */
     void *pContext)

DESCRIPTION
This routine registers a Host Controller Driver with the USB Stack.

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

ERRNO
None

SEE ALSO
usbd

usbHstResetDevice( )

NAME
usbHstResetDevice() – called by the class drivers to reset its device

SYNOPSIS
USBHST_STATUS usbHstResetDevice
    (UINT32 hDevice /* Device handle to be suspended */
    )

DESCRIPTION
This routine is called by the class drivers to reset its device.

RETURNS
USBHST_INVALID_PARAMETERS, USBHST_SUCCESS if Hub driver is called successfully to
reset the device

ERRNO
None

SEE ALSO
device
**usbHstSelectiveResume( )**

**NAME**
usbHstSelectiveResume( ) – resume device.

**SYNOPSIS**
USBHST_STATUS usbHstSelectiveResume

```c
(UINT32 hDevice /* Device handle to be suspended */
)
```

**DESCRIPTION**
This routine is called by the class drivers to selectively resume device.

**RETURNS**
USBHST_INVALID_PARAMETERS, USBHST_SUCCESS if Hub driver is informed about selective resume of the device.

**ERRNO**
None

**SEE ALSO**
device

**usbHstSelectiveSuspend( )**

**NAME**
usbHstSelectiveSuspend( ) – suspend device.

**SYNOPSIS**
USBHST_STATUS usbHstSelectiveSuspend

```c
(UINT32 hDevice /* Device handle to be suspended */
)
```

**DESCRIPTION**
This routine is called by the class drivers to selectively suspend device.

**RETURNS**
USBHST_INVALID_PARAMETERS, USBHST_SUCCESS if Hub driver is informed about selective suspend of the device.

**ERRNO**
None

**SEE ALSO**
device
usbHstSetBitRate()

NAME
usbHstSetBitRate() – set USB bit rate

SYNOPSIS
USBHST_STATUS usbHstSetBitRate(
    UINT32 hDevice,             /* Device handle                         */
    BOOL   bIncrement,          /* Increment (True) or Decrement (False) */
    UINT32 *pCurrentFrameWidth  /* Ptr to current bit rate               */
)

DESCRIPTION
This function is called to set the bit rate of specified USB bus

RETURNS
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, if Parameters are not valid

ERRNO
None

SEE ALSO
urb

usbHstSetConfiguration()

NAME
usbHstSetConfiguration() – set configuration USB request

SYNOPSIS
USBHST_STATUS usbHstSetConfiguration(
    UINT32 hDevice,  /* USB Device handle   */
    UINT16 uIndex    /* Configuration index */
)

DESCRIPTION
This function is called from the class driver layer to issue the SetConfiguration USB Standard Request

RETURNS
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, USBHST_INSUFFICIENT_MEMORY, USBHST_TIMEOUT if request submitted to Host controller Driver is timed out

ERRNO
None

SEE ALSO
urb
**usbHstSetDescriptor()**

**NAME**

`usbHstSetDescriptor()` – set descriptor USB request

**SYNOPSIS**

```c
USBHST_STATUS usbHstSetDescriptor
{
    UINT32 hDevice,  /* USB Device handle */
    UINT8 uDescType, /* USB Descriptor type */
    UINT8 uDescIndex, /* USB Descriptor index */
    UINT16 swLangID,  /* String descriptor language ID */
    UCHAR *pBuffer,  /* Ptr to data buffer */
    UINT32 uSize  /* Num bytes of descriptor to fetch */
}
```

**DESCRIPTION**

This function is called from the class driver layer to issue the SetDescriptor USB Standard Request

**RETURNS**

`USBHST_SUCCESS`, `USBHST_INVALID_PARAMETERS`, `USBHST_INSUFFICIENT_MEMORY`, `USBHST_TIMEOUT` if request submitted to Host controller Driver is timed out

**ERRNO**

None

**SEE ALSO**

`urb`

---

**usbHstSetFeature()**

**NAME**

`usbHstSetFeature()` – set feature USB request

**SYNOPSIS**

```c
USBHST_STATUS usbHstSetFeature
{
    UINT32 hDevice,  /* USB Device handle */
    UINT8 uRecipient, /* Desired recipient */
    UINT16 uIndex,  /* Recipient info list index */
    UINT16 uFeature, /* Feature selector */
    UINT8 uTestSelector /* Test mode selector */
}
```

**DESCRIPTION**

This function is called by the class driver layer to issue the SetFeature USB Standard Request

**RETURNS**

`USBHST_SUCCESS`, `USBHST_INVALID_PARAMETERS`, `USBHST_INSUFFICIENT_MEMORY`, `USBHST_TIMEOUT` if request submitted to Host controller Driver is timed out

**ERRNO**

None

**SEE ALSO**

`urb`
**usbHstSetInterface( )**

**NAME**

`usbHstSetInterface( )` – set interface USB request

**SYNOPSIS**

```c
USBHST_STATUS usbHstSetInterface
  (  
    UINT32 hDevice, /* USB device handle */
    UINT16 uIndex, /* Interface index */
    UINT16 uAltIndex /* Alternate interface index */
  )
```

**DESCRIPTION**

This routine is called from the class driver layer to issue the SetInterface USB Standard Request

**RETURNS**

`USBHST_SUCCESS`, `USBHST_INVALID_PARAMETERS`, `USBHST_INSUFFICIENT_MEMORY`, `USBHST_TIMEOUT` if request submitted to Host controller Driver is timed out

**ERRNO**

None

**SEE ALSO**

`urb`

---

**usbHstSetSynchFrame( )**

**NAME**

`usbHstSetSynchFrame( )` – set synch frame USB request

**SYNOPSIS**

```c
USBHST_STATUS usbHstSetSynchFrame
  (  
    UINT32 hDevice, /* USB Device handle */
    UINT16 uEndPointNumber, /* Endpoint number */
    UCHAR *pBuffer /* Ptr to buffer to hold synch frame number */
  )
```

**DESCRIPTION**

This function is called by the class driver layer to issue the SetSynchFrame USB Standard Request

**RETURNS**

`USBHST_SUCCESS`, `USBHST_INVALID_PARAMETERS`, `USBHST_INSUFFICIENT_MEMORY`, `USBHST_TIMEOUT` if request submitted to Host controller Driver is timed out

**ERRNO**

None

**SEE ALSO**

`urb`
**usbHstURBCancel()**

**NAME**  
usbHstURBCancel() – cancel USB Request Block

**SYNOPSIS**  
```c
USBHST_STATUS usbHstURBCancel
    (pUSBHST_URB pURB /* Ptr to USB Request Block */)
```

**DESCRIPTION**  
This function is called by the class driver layer to cancel the request submitted to HCD

**RETURNS**  
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, if Parameters are not valid

**ERRNO**  
None

**SEE ALSO**  
urb

**usbHstURBSubmit()**

**NAME**  
usbHstURBSubmit() – submit USB request block

**SYNOPSIS**  
```c
USBHST_STATUS usbHstURBSubmit
    (pUSBHST_URB pURB /* Ptr to USB Request Block */)
```

**DESCRIPTION**  
This function is called by the class driver layer to submit a USB Request Block to the underlying HCD (either OHCD, UHCD, or EHCD)

**RETURNS**  
USBHST_SUCCESS, USBHST_INVALID_PARAMETERS, if Parameters are not valid

**ERRNO**  
None

**SEE ALSO**  
urb
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

usbInit( )

**NAME**
usbInit() – initialize the USB2 stack

**SYNOPSIS**
STATUS usbInit (void)

**DESCRIPTION**
This function initializes the USB2 Stack
usbKbdDevCreate( )

NAME  
usbKbdDevCreate( ) – create a VxWorks device for an USB keyboard

SYNOPSIS  
STATUS usbKbdDevCreate
        (char    * name,     /* name to use for this device      */
         SIO_CHAN * pSioChan  /* pointer to core driver structure */)

DESCRIPTION  
This routine creates a device on a specified serial channel. Each channel to be used should
have exactly one device associated with it by calling this routine.

For instance, to create the device "/ /0", the proper call would be:
        usbKbdDevCreate ("/usbKb/0", pSioChan);

Where pSioChan is the address of the underlying SIO_CHAN serial channel descriptor
(defined in sioLib.h). This routine is typically called by the USB keyboard driver, when it
detects an insertion of a USB keyboard.

RETURNS  
OK, or ERROR if the driver is not installed, or the device already exists, or failed to allocate
memory.

ERRNO  
none

SEE ALSO  
usrUsbKbdInit
usbKbdDrvUnInit()

NAME       usbKbdDrvUnInit() – shuts down an I/O USB keyboard driver
SYNOPSIS   STATUS usbKbdDrvUnInit (void)
DESCRIPTION This is supplied to for the user, but it should be noted that iosDrvRemove() may cause unpredictable results.
RETURNS    OK or ERROR
ERRNO      none
SEE ALSO   usrUsbKbdInit

usbKeyboardDevInit()

NAME       usbKeyboardDevInit() – initialize USB keyboard SIO driver
SYNOPSIS   STATUS usbKeyboardDevInit (void)
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 USBD Layer
SEE ALSO   usbKeyboardLib
## usbKeyboardDevShutdown()

### NAME

`usbKeyboardDevShutdown()` – shuts down keyboard SIO driver

### SYNOPSIS

```c
STATUS usbKeyboardDevShutdown (void)
```

### DESCRIPTION

This function shuts down the keyboard driver. The driver is shutdown only if `initCount` after decrementing. If it is more than 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

```c
STATUS usbKeyboardDynamicAttachRegister
   (USB_KBD_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:

```c
typedef (*USB_KBD_ATTACH_CALLBACK)
   (pVOID arg,
    SIO_CHAN *pSioChan,
    UINT16 attachCode
   )
```

`usbKeyboardLib` will invoke `callback` each time a USB keyboard 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 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`.

### RETURNS

`OK`, or `ERROR` if unable to register callback
ERRNO

S_usbKeyboardLib_BAD_PARAM
Bad Parameter are passed

S_usbKeyboardLib_OUT_OF_MEMORY
Not sufficient memory is available

SEE ALSO

usbKeyboardLib

usbKeyboardDynamicAttachUnregister( )

NAME

usbKeyboardDynamicAttachUnregister( ) – Unregisters keyboard attach callback

SYNOPSIS

STATUS usbKeyboardDynamicAttachUnRegister

(   USB_KBD_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 keyboard attachment
and removal. The callback and arg parameters must exactly match those passed in a previous
call to usbKeyboardDynamicAttachRegister().

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**

```c
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**

```c
pVOID usbListFirst
(  pLIST_HEAD pListHead  /* head of linked list */   )
```

**DESCRIPTION**

Returns the pointer to the first structure in a linked list given a pointer to the `LIST_HEAD`.

**RETURNS**

`pStruct` of first structure on list or `NULL` if list empty.

**ERRNO**

None.

**SEE ALSO**

`usbListLib`

---

**usbListLink()**

**NAME**

`usbListLink()` – Add an element to a linked list

**SYNOPSIS**

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

**DESCRIPTION**

Using the LINK structure `pLink`, add `pStruct` to the list of which the list head is `pHead`. `flag` must be `LINK_HEAD` or `LINK_TAIL`.

**RETURNS**

N/A

**ERRNO**

None.

**SEE ALSO**

`usbListLib`
**usbListLinkProt( )**

**NAME**
usbListLinkProt( ) – Add an element to a list guarded by a mutex

**SYNOPSIS**
```c
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 */
     )
```

**DESCRIPTION**
This function is similar to `linkList()` with the addition that this function will take the `mutex` prior to manipulating the list.

**NOTE**
The function will block forever if the mutex does not become available.

**RETURNS**
N/A

**ERRNO**
None.

**SEE ALSO**
usbListLib

---

**usbListNext( )**

**NAME**
usbListNext( ) – Retrieves next pStruct in a linked list

**SYNOPSIS**
```c
pVOID usbListNext
     ( 
       pLINK pLink  /* LINK structure */
     )
```

**DESCRIPTION**
Returns the pointer to the next structure in a linked list given a `pLink` pointer. The value returned is the `pStruct` of the element in the linked list which follows the current `pLink`, not a pointer to the following `pLink`. (Typically, a client is more interested in walking its own list of structures rather than the LINK structures used to maintain the linked list.

**RETURNS**
pStruct of next structure in list or NULL if end of list.

**ERRNO**
None.

**SEE ALSO**
usbListLib
usbListUnlink( )

NAME   usbListUnlink() – Remove an entry from a linked list

SYNOPSIS VOID usbListUnlink
             (pLINK pLink  /* LINK structure to be unlinked */)

DESCRIPTION Removes pLink 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 VOID usbListUnlinkProt
             (pLINK        pLink,  /* LINK structure to be unlinked */
              MUTEX_HANDLE mutex   /* list guard mutex */)

DESCRIPTION This function is the same as usbListUnlink() with the addition that this function will take
the mutex prior to manipulating the list.

NOTE The function will block forever if the mutex does not become available.

RETURNS N/A

ERRNO None.

SEE ALSO usbListLib
### usbLoadPegasus()

**NAME**

usbLoadPegasus() – load (create) USB END device

**SYNOPSIS**

```c
STATUS usbLoadPegasus
    (int unitNum,
     USB_PEGASUS_DEV * pDev)
```

**DESCRIPTION**

This function is taken from usrEndLibInit() and modified suitably to load the end driver when USB Ethernet device is dynamically attached or detached.

**RETURNS**

OK or ERROR.

**ERRNO**

none

**SEE ALSO**

usrUsbPegasusEndInit

---

### usbLogMsg()

**NAME**

usbLogMsg() – Dumps the log messages

**SYNOPSIS**

```c
void usbLogMsg
    (char * msg,
     int a1,
     int a2,
     int a3,
     int a4,
     int a5,
     int a6)
```

**DESCRIPTION**

This function dumps the content of the log messages. Before dumping the contents it checks whether the INCLUDE_LOGGING is defined.

**RETURNS**

N/A

**ERRNO**

none

**SEE ALSO**

usrUsbInit
### usbMemToPci()

**NAME**
usbMemToPci() – Convert a memory address to a PCI-reachable memory address

**SYNOPSIS**
```c
UINT32 usbMemToPci
    (pVOID pMem /* memory address to convert */)
```

**DESCRIPTION**
Converts `pMem` to an equivalent 32-bit memory address visible from the PCI bus. This conversion is necessary to allow PCI bus masters to address the same memory viewed by the processor.

**RETURNS**
converted memory address

**ERRNO**
Not Available

**SEE ALSO**
usbPciStub

---

### usbMouseDevInit()

**NAME**
usbMouseDevInit() – initialize USB mouse SIO driver

**SYNOPSIS**
```c
STATUS usbMouseDevInit (void)
```

**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 USBD Layer

**SEE ALSO**
usbMouseLib
usbMouseDevShutdown()

NAME  usbMouseDevShutdown() – shuts down mouse SIO driver
SYNOPSIS STATUS usbMouseDevShutdown (void)
DESCRIPTION This function shuts down the mouse SIO driver. If after decrementing \texttt{initCount} is 0, SIO driver is uninitialized.
RETURNS OK, or \texttt{ERROR} if unable to shutdown.
ERRNO S_usbMouseLib\_NOT\_INITIALIZED SIO Driver is not initialized
SEE ALSO usbMouseLib

usbMouseDynamicAttachRegister()

NAME  usbMouseDynamicAttachRegister() – Register mouse attach callback
SYNOPSIS STATUS usbMouseDynamicAttachRegister
  {  \texttt{USB\_MSE\_ATTACH\_CALLBACK} \texttt{callback},  /* new callback to be registered */  \texttt{pVOID} \texttt{arg}  /* user-defined arg to callback */  }
DESCRIPTION \texttt{callback} is a caller-supplied function of the form:
\begin{verbatim}
typedef (*\texttt{USB\_MSE\_ATTACH\_CALLBACK})
  {
    \texttt{pVOID} \texttt{arg},
    \texttt{SIO\_CHAN} *\texttt{pSioChan},
    \texttt{UINT16} \texttt{attachCode}
  };
\end{verbatim}
\texttt{usbMouseLib} will invoke \texttt{callback} each time a USB mouse is attached to or removed from the system. \texttt{arg} is a caller-defined parameter which will be passed to the \texttt{callback} each time it is invoked. The \texttt{callback} will also be passed a pointer to the \texttt{SIO\_CHAN} structure for the channel being created/destroyed and an attach code of \texttt{USB\_MSE\_ATTACH} or \texttt{USB\_MSE\_REMOVE}.
RETURNS OK, or \texttt{ERROR} if unable to register \texttt{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

STATUS usbMouseDynamicAttachUnRegister

(    USB_MSE_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 mouse attachment and removal. The callback and arg parameters 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

STATUS usbMouseSioChanLock

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

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**  
```c
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

void usbMsBulkInErpInUseFlagSet

(  
    BOOL state

)

DESCRIPTION

This function is used to set the state of Bulk-IN ERP flag. state is the state to set.

RETURNS

N/A

ERRNO

none

SEE ALSO

usbTargMsLib
usbMsBulkInErpInit()

NAME        usbMsBulkInErpInit() – initialize the bulk-in ERP
SYNOPSIS    STATUS usbMsBulkInErpInit
             (UINT8  * pData,        /* pointer to data */
              UINT32  size,         /* size of data */
              ERP_CALLBACK erpCallback,  /* erp callback */
              pVOID         usrPtr        /* user pointer */)
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 in use flag

SYNOPSIS
BOOL usbMsBulkOutErpInUseFlagGet (void)

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
void usbMsBulkOutErpInUseFlagSet (  
    BOOL state  /* State to set */  
)

DESCRIPTION
This function is used to set the state of Bulk - OUT ERP flag. state is the state to set.

RETURNS
N/A

ERRNO
none

SEE ALSO
usbTargMsLib

usbMsBulkOutErpInit()

NAME
usbMsBulkOutErpInit( ) – initialize the bulk-Out ERP

SYNOPSIS
STATUS usbMsBulkOutErpInit  
    (  
        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 if 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 unstalls 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
2 Routines

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   void usbMsTestRxCallback
            (pVOID p)
DESCRIPTION This function is invoked after the Bulk OUT test data is transmitted. It sets the bulk OUT flag to false.
RETURNS    N/A
ERRNO       N/A
SEE ALSO    usbTargMsLib
#### usbMsTestTxCallback()

**NAME**
usbMsTestTxCallback() – invoked after test data transmitted

**SYNOPSIS**
```c
void usbMsTestTxCallback
```(pVOID p)

**DESCRIPTION**
This function is invoked after the Bulk IN test data is transmitted. It sets the bulk IN flag to false.

**RETURNS**
N/A

**ERRNO**
one

**SEE ALSO**
usbTargMsLib

---

#### usbMseDrvUnInit()

**NAME**
usbMseDrvUnInit() – shuts down an I/O USB mouse driver

**SYNOPSIS**
```c
STATUS usbMseDrvUnInit (void)
```

**DESCRIPTION**
This is supplied to the user, but it should be noted that iosDrvRemove() may cause unpredictable results.

**RETURNS**
OK or ERRNO, if not able to shutdown the USB Mouse Driver

**ERRNO**
one

**SEE ALSO**
usrUsbMseInit

---

#### usbOhciDumpEndpointDescriptor()

**NAME**
usbOhciDumpEndpointDescriptor() – dump endpoint descriptor contents

**SYNOPSIS**
```c
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
VOID usbOhciDumpGeneralTransferDescriptor
(   PVOID pGeneralTransferDescriptor
)

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

PARAMETERS
pGeneralTransferDescriptor (IN) - Pointer to the general descriptor

RETURNS
N/A

ERRNO
None.

SEE ALSO
usbOhciDebug

usbOhciDumpMemory()

NAME
usbOhciDumpMemory() – dump memory contents

SYNOPSIS
VOID usbOhciDumpMemory
(   UINT32 uAddress,
   UINT32 uLength,
   PVOID pGeneralTransferDescriptor
)
USB Ohci Dump Pending Transfers

DESCRIPTION
This function is used to dump the pending transfers for the endpoint.

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

RETURNS
N/A

ERRNO
None.

SEE ALSO
- usbOhciDebug

USB Ohci Dump Pending Transfers

NAME
usbOhciDumpPendingTransfers() – dump pending transfers

SYNOPSIS
VOID usbOhciDumpPendingTransfers
{
    PVOID pEndpointDescriptor
}

DESCRIPTION
This function is used to dump the contents of the specified memory location.

PARAMETERS
- *uAddress* (IN) - Specifies the address of memory location.
- *uLength* (IN) - Specifies the length of memory to be dumped.
- *uWidth* (IN) - 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

USB Ohci Dump Periodic Endpoint List

NAME
usbOhciDumpPeriodicEndpointList() – dump periodic endpoint descriptor list

SYNOPSIS

DESCRIPTION

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

RETURNS
N/A

ERRNO
None.

SEE ALSO
- usbOhciDebug
SYNOPSIS
VOID usbOhciDumpPeriodicEndpointList
    (UINT8 uHostControllerIndex)

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

PARAMETERS
uHostControllerIndex (IN) - Specifies the host controller index

RETURNS
N/A

ERRNO
None.

SEE ALSO
usbOhciDebug

usbOhciDumpRegisters()

NAME
usbOhciDumpRegisters() – dump registers contents.

SYNOPSIS
BOOLEAN usbOhciDumpRegisters
    (UINT32 uHostControllerIndex)

DESCRIPTION
This function is used to dump the contents of the USB OHCI Host Controller Registers.

PARAMETERS
uHostControllerIndex (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() – uninitialise the USB OHCI Host Controller Driver.

SYNOPSIS
BOOLEAN usbOhciExit (void)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Returns</th>
<th>See Also</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>usbOhciInit()</code></td>
<td>This function uninitialises the OHCI Host Controller Driver.</td>
<td><code>FALSE, TRUE</code> if all the OHCI Host Controllers are reset and the cleanup is successful.</td>
<td><code>usbOhci</code></td>
</tr>
<tr>
<td><code>usbOhciInitializeModuleTestingFunctions()</code></td>
<td>Function to obtain the entry points used for HCD module testing.</td>
<td><code>N/A</code></td>
<td><code>usbOhciDebug</code></td>
</tr>
</tbody>
</table>

**usbOhciInit()**

- **NAME**: `usbOhciInit()` – initialise the USB OHCI Host Controller Driver.
- **SYNOPSIS**: `BOOLEAN usbOhciInit (void)`
- **DESCRIPTION**: This function initialises the OHCI Host Controller Driver. It detects the number of USB OHCI Controllers present on the system and initializes all the OHCI controllers.
- **PARAMETERS**: N/A
- **RETURNS**: `TRUE` if the OHCI Host Controllers are initialized, otherwise `FALSE`
- **ERRNO**: None.
- **SEE ALSO**: `usbOhci`

**usbOhciInitializeModuleTestingFunctions()**

- **NAME**: `usbOhciInitializeModuleTestingFunctions()` – obtains entry points
- **SYNOPSIS**: `VOID usbOhciInitializeModuleTestingFunctions` 
  ```c
  ( 
    PUSBHST_HC_DRIVER_TEST pHCDriverTestEntryPoints /* Ptr to HCD module entry points */ 
  )
  ```
- **DESCRIPTION**: Function to obtain the entry points used for HCD module testing.
- **RETURNS**: N/A
- **ERRNO**: None
- **SEE ALSO**: `usbOhciDebug`
usbPciByteGet()

NAME
usbPciByteGet() – Returns a UINT8 configuration value

SYNOPSIS
UINT8 usbPciByteGet
{
    UINT8 busNo, /* Bus number of device */
    UINT8 deviceNo, /* Device number of device */
    UINT8 funcNo, /* Function number of device */
    UINT16 regOffset /* Offset into PCI config space */
}

DESCRIPTION
This function returns the UINT8 value at offset regOffset from the PCI configuration space of the device identified by busNo, deviceNo, and funcNo.

RETURNS
UINT8 value read from device configuration space

ERRNO
Not Available

SEE ALSO
usbPciStub

usbPciByteIn()

NAME
usbPciByteIn() – input a byte from PCI I/O space

SYNOPSIS
UINT8 usbPciByteIn
{
    UINT32 address /* PCI I/O address */
}

DESCRIPTION
Inputs a byte from a PCI I/O address address.

RETURNS
byte input from i/o address

ERRNO
Not Available

SEE ALSO
usbPciStub
usbPciByteOut()

NAME
usbPciByteOut() – output a byte to PCI I/O space

SYNOPSIS
VOID usbPciByteOut
  (
    UINT32 address, /* PCI I/O address */
    UINT8  value    /* value */
  )

DESCRIPTION
Outputs value to the PCI I/O address address.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usbPciStub

usbPciClassFind()

NAME
usbPciClassFind() – Locates PCI devices by class.

SYNOPSIS
BOOL usbPciClassFind
  (
    UINT8  pciClass,    /* PCI device class */
    UINT8  subClass,    /* PCI device sub-class */
    UINT8  pgmIf,       /* Programming interface */
    UINT16 index,      /* Caller wants nth matching dev */
    pUINT8 pBusNo,     /* Bus number of matching dev */
    pUINT8 pDeviceNo,  /* Device number of matching dev */
    pUINT8 pFuncNo     /* Function number of matching dev */
  )

DESCRIPTION
A caller uses this function to locate a PCI device by its PCI class. The caller must specify the
pciClass, subClass, and pgmIf for the device being sought. The function returns the first
matching device for index = 0, the second for index = 1, and so forth. The bus number, device
number, and function number for the matching device are returned in the pBusNo,
pDeviceNo, and pFuncNo buffers provided by the caller.

RETURNS
Returns TRUE if matching device found, FALSE if device not found.

ERRNO
Not Available

SEE ALSO
usbPciClassFind
usbPciConfigHeaderGet()

NAME
usbPciConfigHeaderGet() – Reads a device’s PCI configuration header

SYNOPSIS
VOID usbPciConfigHeaderGet
{
    UINT8    busNo,     /* Bus number of device */
    UINT8    deviceNo,  /* Device number of device */
    UINT8    funcNo,    /* Function number of device */
    pPCI_CFG_HEADER pCfgHdr    /* Buffer provided by caller */
}

DESCRIPTION
This function reads the PCI configuration header for the device identified by busNo, deviceNo, and funcNo. The configuration header is stored in the pCfgHdr buffer provided by the caller.

This function initializes the pCfgHdr structure to zeros. Any fields which cannot be read from the device’s configuration header will remain zero upon return. This function does not attempt to read fields defined as “reserved” in the PCI configuration header.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usbPciStub

usbPciDwordGet()

NAME
usbPciDwordGet() – Returns a UINT32 configuration value

SYNOPSIS
UINT32 usbPciDwordGet
{
    UINT8    busNo,     /* Bus number of device */
    UINT8    deviceNo,  /* Device number of device */
    UINT8    funcNo,    /* Function number of device */
    UINT16   regOffset  /* Offset into PCI config space */
}

DESCRIPTION
This function returns the UINT32 value at offset regOffset from the PCI configuration space of the device identified by busNo, deviceNo, and funcNo.

NOTE
This function adjusts for big vs. little endian environments.

RETURNS
UINT32 value read from device configuration space

ERRNO
Not Available

SEE ALSO
usbPciStub
usbPciDwordIn()

NAME
usbPciDwordIn() – input a dword from PCI I/O space

SYNOPSIS
UINT32 usbPciDwordIn
{
    UINT32 address /* PCI I/O address */
}

DESCRIPTION
Inputs a dword from a PCI I/O address address.

NOTE
This function adjusts for big vs. little endian environments.

RETURNS
dword input from i/o address

ERRNO
Not Available

SEE ALSO
usbPciStub

usbPciDwordOut()

NAME
usbPciDwordOut() – outputs a dword to PCI I/O space

SYNOPSIS
VOID usbPciDwordOut
{
    UINT32 address, /* PCI I/O address */
    UINT32 value /* value */
}

DESCRIPTION
Outputs value to the PCI I/O address address.

NOTE
This function adjusts for big vs. little endian environments.

RETURNS
N/A

ERRNO
Not Available

SEE ALSO
usbPciStub
usbPciIntConnect()

NAME  
usbPciIntConnect() – Connect to a interrupt vector

SYNOPSIS  
STATUS usbPciIntConnect
{  
    INT_HANDLER_PROTOTYPE func, /* new interrupt handler */  
    pVOID param,  /* parameter for int handler */  
    UINT16 intNo /* interrupt vector number */  
}

DESCRIPTION  
Connects the func to the interrupt number intNo. param is an application-specific value which will be passed to func each time the interrupt handler is invoked.

RETURNS  
OK, or ERROR if unable to connect/enable interrupt

ERRNO  
Not Available

SEE ALSO  
usbPciStub

usbPciIntDisconnect()

NAME  
usbPciIntDisconnect() – Removes an interrupt handler

SYNOPSIS  
VOID usbPciIntRestore
{  
    INT_HANDLER_PROTOTYPE func, /* int handler to be removed */  
    pVOID param,  /* parameter for int handler */  
    UINT16 intNo /* interrupt vector number */  
}

DESCRIPTION  
Removes an interrupt handler installed by usbPciIntConnect(). func, param, and intNo must match the corresponding parameters from an earlier call to usbPciIntConnect().

RETURNS  
N/A

ERRNO  
Not Available

SEE ALSO  
usbPciStub
usbPciMemFlush()

NAME       usbPciMemFlush() – Flush a region of memory through the cache

SYNOPSIS   VOID usbPciMemFlush
            (pVOID  pMem,  /* base of memory region to invalidate */
             UINT32 size   /* size of region to invalidate */)

DESCRIPTION In systems which implement a non-write-thru cache, the processor may have written data
to memory which has not yet been flushed to the actual system memory. Before other bus
masters may interrogate this memory, it may be necessary to flush the cache.

NOTE       Returns immediately if size == 0.

RETURNS    N/A

ERRNO      Not Available

SEE ALSO   usbPciStub

usbPciMemInvalidate()

NAME       usbPciMemInvalidate() – Invalidate cache for a region of memory

SYNOPSIS   VOID usbPciMemInvalidate
            (pVOID  pMem,  /* base of memory region to invalidate */
             UINT32 size   /* size of region to invalidate */)

DESCRIPTION When another bus master, such as a PCI bus master, writes to memory, the cache may need
to be invalidated for that region of memory.

NOTE       Returns immediately if size == 0.

RETURNS    N/A

ERRNO      Not Available

SEE ALSO   usbPciStub
### usbPciMemioOffset()

**NAME**
usbPciMemioOffset() – Return PCI MEMIO to CPU MEMIO offset

**SYNOPSIS**

```c
UINT32 usbPciMemioOffset (void)
```

**DESCRIPTION**
For memory-mapped I/O, the CPU’s view of a memory address may not be the same as that programmed into the base address register of a PCI adapter. The CPU should add the value returned by this function to the BAR in order to produce the correct CPU-visible memory address.

**RETURNS**
USB_PCI_MEMIO_OFFSET

**ERRNO**
Not Available

**SEE ALSO**
usbPciStub

### usbPciToMem()

**NAME**
usbPciToMem() – Convert a PCI-reachable address to a CPU-reachable pointer

**SYNOPSIS**

```c
pVOID usbPciToMem
    (UINT32 pciAdrs /* 32-bit PCI address to be converted */)
```

**DESCRIPTION**
Converts `pciAdrs` to an equivalent CPU memory address.

**RETURNS**
pointer to PCI memory

**ERRNO**
Not Available

**SEE ALSO**
usbPciStub
usbPciWordGet()

NAME
usbPciWordGet() – Returns a UINT16 configuration value

SYNOPSIS
UINT32 usbPciWordGet
(  
UINT8  busNo,     /* Bus number of device */
UINT8  deviceNo,  /* Device number of device */
UINT8  funcNo,    /* Function number of device */
UINT16 regOffset  /* Offset into PCI config space */
)

DESCRIPTION
This function returns the UINT16 value at offset regOffset from the PCI configuration space of the device identified by busNo, deviceNo, and funcNo.

NOTE
This function adjusts for big vs. little endian environments.

RETURNS
UINT16 value read from device configuration space

ERRNO
Not Available

SEE ALSO
usbPciStub

usbPciWordIn()

NAME
usbPciWordIn() – input a word from PCI I/O space

SYNOPSIS
UINT16 usbPciWordIn
(  
 UINT32 address  /* PCI I/O address */
)

DESCRIPTION
Inputs a word from a PCI I/O address address.

NOTE
This function adjusts for big vs. little endian environments.

RETURNS
word input from i/o address

ERRNO
Not Available

SEE ALSO
usbPciStub
usbPciWordOut()

NAME       usbPciWordOut() – outputs a word to PCI I/O space

SYNOPSIS   VOID usbPciWordOut
            {
                UINT32 address, /* PCI I/O address */
                UINT16 value    /* value */
            }

DESCRIPTION Outputs value to the PCI I/O address address.

NOTE       This function adjusts for big vs. little endian environments.

RETURNS     N/A

ERRNO       Not Available

SEE ALSO   usbPciStub

usbPegasusAttachCallback()

NAME       usbPegasusAttachCallback() – configuration level callback

SYNOPSIS   VOID usbPegasusAttachCallback
            {
                pVOID arg, /* caller-defined argument */
                USB_PEGASUS_DEV * pDev, /* pointer to Pegasus Device */
                UINT16 attachCode /* attach code */
            }

DESCRIPTION When a Pegasus device is inserted, this will get called. It passes the attach message to tUsbPgs task.

RETURNS     N/A

ERRNO       none

SEE ALSO   usrUsbPegasusEndInit
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 a 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**

```c
STATUS usbPegasusDynamicAttachRegister(  
    USB_PEGASUS_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:

```c
typedef (*USB_PEGASUS_ATTACH_CALLBACK)(
    pVOID arg,
    USB_PEGASUS_DEV * pDev,
    UINT16 attachCode
);
```

usbPegasusDevLib will invoke callback each time a PEGASUS 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 pass the structure of the device being created/destroyed and an attach code of USB_PEGASUS_ATTACH or USB_PEGASUS_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 USBD 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 callback

**SYNOPSIS**  
STATUS usbPegasusDynamicAttachUnregister  
(  
  USB_PEGASUS_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 attachment and removal. The callback and arg parameters 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 register the attachment callback

**SEE ALSO**  
usbPegasusEnd

---

**usbPegasusEndInit( )**

**NAME**  
usbPegasusEndInit( ) – initializes the pegasus library

**SYNOPSIS**  
STATUS usbPegasusEndInit(void)

**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 reegisters 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

initString

The device initialization string.

RETURNS

An END object pointer or NULL on error.

ERRNO

none

SEE ALSO

usbPegasusEnd
usbPegasusEndStart()

NAME
usbPegasusEndStart() – make the network device operational

SYNOPSIS
void usbPegasusEndStart
   (int    unitNum,      /* unit number */
    char * pAddrString,  /* enet address */
    int    netmask       /* netmask */
   )

DESCRIPTION
After muxDevStart has been executed, this then attaches an IP address to the USB device and then starts the it.

RETURNS
N/A

ERRNO
none

SEE ALSO
usrUsbPegasusEndInit

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. Evertime the function is called the global initCount will be decremented. The driver will be truely un-initialized only when initCount 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

usbPeriLogMsg()

NAME
usbPeriLogMsg() – dumps the log messages

SYNOPSIS
void usbPeriLogMsg
    (char * msg,
     int   a1,
     int   a2,
     int   a3,
     int   a4,
     int   a5,
     int   a6
    )

DESCRIPTION
This function dumps the content of the log messages. Before dumping the contents it checks whether the INCLUDE_LOGGING is defined.

RETURNS
N/A

ERRNO
none

SEE ALSO
usrUsbTargInit
**usbPeriphWindViewLog( )**

**NAME**
usbPeriphWindViewLog( ) – USB2.0 Peripheral Stack WindView Log Events

**SYNOPSIS**
```c
VOID usbPeriphWindViewLog
    (UINT32 evId,
     char   *buffer,
     UINT32 mask)
```

**DESCRIPTION**
The USB2 Peripheral Stack is instrumented to provide additional WindView User log events for function entries/entries tracing within the Peripheral Stack. Logging is based on the Filter value has been initialized with the appropriate filter mask. If WindView is not enabled and the correct filter mask is set, then the WindView user event message will be logged to the standard output.

```c
#define USB_TARG_WV_FILTER    0x00000001
#define USB_HAL_WV_FILTER     0x00000002
#define USB_TCD_ISP582_WV_FILTER 0x00000004
#define USB_TCD_PDIUSBD12_WV_FILTER 0x00000008
```

**RETURNS**
N/A

**ERRNO**
none

**SEE ALSO**
usrUsbTargInit

---

**usbPrinterDevInit( )**

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

**SYNOPSIS**
```c
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_USBDA_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  
STATUS usbPrinterDynamicAttachRegister  
(  
    USB_PRN_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_PRN_ATTACH_CALLBACK)  
(  
    pVOID arg,  
    SIO_CHAN *pSioChan,  
    UINT16 attachCode  
)

usbPrinterLib will invoke callback each time a USB printer 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 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.

RETURNS  
OK, or ERROR if unable to register callback

ERRNO  
S_usbPrinterLib_BAD_PARAM  
Bad Parameters received
S_usbPrinterLib_OUT_OF_MEMORY
  System out of memory

SEE ALSO   usbPrinterLib

---

**usbPrinterDynamicAttachUnregister( )**

**NAME**

usbPrinterDynamicAttachUnregister( ) – Unregisters printer attach callback

**SYNOPSIS**

```c
STATUS usbPrinterDynamicAttachUnRegister
  (USB_PRN_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 printer attachment
and removal. The `callback` and `arg` parameters 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**

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

**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.
usbPrinterSioChanUnlock()

NAME

usbPrinterSioChanUnlock() – Marks SIO_CHAN structure as unused

SYNOPSIS

STATUS usbPrinterSioChanUnlock  
  (  
   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 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 unclock

SEE ALSO

usbPrinterLib

usbPrnDrvUnInit()

NAME

usbPrnDrvUnInit() – shuts down an I/O USB printer driver

SYNOPSIS

STATUS usbPrnDrvUnInit (void)

DESCRIPTION

This is supplied to for the user, but it should be noted that iosDrvRemove() may cause unpredictable results.
**usbPrnWrTest()**

**NAME**

`usbPrnWrTest()` – test the USB printer write function

**SYNOPSIS**

```c
STATUS usbPrnWrTest
    (char    * deviceName,  /* Printer device name */
     char    * fileName     /* file to print */)
```

**DESCRIPTION**

This is given as a sample application to print a file using the i/o system calls to access the printer device. It has been assumed that the file that it sent to the printer has been "canned" for the particular printer being used. Refer to the USB Developer's Kit manual for further details.

**RETURNS**

OK, or ERROR.

**ERRNO**

none

**SEE ALSO**

`usrUsbPrnInit`

---

**usbQueueCreate()**

**NAME**

`usbQueueCreate()` – Creates a O/S-independent queue structure

**SYNOPSIS**

```c
STATUS usbQueueCreate
    (UINT16        depth,        /* Max entries queue can handle */
     pQUEUE_HANDLE pQueueHandle  /* Handle of newly created queue */)
```

**DESCRIPTION**

This function creates a queue which can accomodate a number of USB_MESSAGE entries according to the `depth` parameter. Returns the `<pQueueHandle>` of the newly created queue if successful.
usbQueueDestroy( )

NAME
usbQueueDestroy( ) – Destroys a queue

SYNOPSIS
STATUS usbQueueDestroy
{
    QUEUE_HANDLE queueHandle /* Handle of queue to destroy */
}

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
STATUS usbQueueGet
{
    QUEUE_HANDLE queueHandle, /* queue handle */
    pUSB_MESSAGE pMsg, /* USB_MESSAGE to receive msg */
    UINT32 blockFlag /* specifies blocking action */
}

DESCRIPTION
Retrieves a message from the specified queueHandle and stores it in pOssMsg. If the queue is empty, blockFlag specifies the blocking behavior. OSS_BLOCK blocks indefinitely, OSS_DONT_BLOCK does not block and returns an error immediately if the queue is empty.
2 Routines

usbQueuePut()

NAME

usbQueuePut( ) – Puts a message onto 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 onto queueHandle. This function will only block 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/isoch 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 (e.g., interrupt or isochronous transfers) an HCD needs to be able to calculate the amount of 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. The 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  

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.
**NAME**

`usbSpeakerDevShutdown()` – shuts down speaker SIO driver

**SYNOPSIS**

```c
STATUS usbSpeakerDevShutdown (void)
```

**DESCRIPTION**

This function shuts down speaker SIO driver depending on `initCount`. Every call to this function decrements the `initCount`, and when it turns 0, SIO speaker driver is shutdown.

**RETURNS**

OK, or ERROR if unable to shutdown.

**ERRNO**

- `S_usbSpeakerLib_OUT_OF_RESOURCES`
  - Sufficient resources not available
- `S_usbSpeakerLib_USBD_FAULT`
  - Fault in the USBD Layer

**SEE ALSO**

`usbSpeakerLib`

---

**NAME**

`usbSpeakerDynamicAttachRegister()` – Register speaker attach callback

**SYNOPSIS**

```c
STATUS usbSpeakerDynamicAttachRegister
    (USB_SPKR_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:

```c
typedef (*USB_SPKR_ATTACH_CALLBACK)
    (pVOID arg,
     SEQ_DEV *pSeqDev,
     UINT16 attachCode
    );
```
**usbSpeakerLib** will invoke *callback* each time a USB speaker 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 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**.

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

**ERRNO**  
S_usbSpeakerLib_BAD_PARAM  
Bad Parameter is received  
S_usbSpeakerLib_OUT_OF_MEMORY  
Sufficient memory is not available

**SEE ALSO**  
usbSpeakerLib

---

**usbSpeakerDynamicAttachUnregister()**

**NAME**  
usbSpeakerDynamicAttachUnregister() – Unregisters speaker attach callback

**SYNOPSIS**  
```c
STATUS usbSpeakerDynamicAttachUnRegister
(  
    USB_SPKR_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 speaker attachment and removal. The *callback* and *arg* parameters must exactly match those passed in a previous call to **usbSpeakerDynamicAttachRegister()**.

**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

usbSpkrDevCreate()

NAME

usbSpkrDevCreate( ) – create a VxWorks device for an USB speaker

SYNOPSIS

STATUS usbSpkrDevCreate
    (char   *         name,    /* name to use for this device      */
     SEQ_DEV        * pSeqDev  /* pointer to core driver structure */)

DESCRIPTION

This routine creates a device on a specified serial channel. Each channel to be used should have exactly one device associated with it by calling this routine.

For instance, to create the device "/usbSp/0", the proper call would be:

    usbSpkrDevCreate ("/usbSp/0", pSeqDev);

Where pSeqDev is the address of the underlying SEQ_DEV serial channel descriptor (defined in sioLib.h). This routine is typically called by the USB speaker driver, when it detects an insertion of a USB speaker.

RETURNS

OK, or ERROR if the driver is not installed, or the device already exists, or failed to allocate memory.

ERRNO

none

SEE ALSO

usrUsbSpkrInit

usbSpkrDrvUnInit()

NAME

usbSpkrDrvUnInit( ) – shuts down an I/O USB speaker driver

SYNOPSIS

STATUS usbSpkrDrvUnInit (void)

DESCRIPTION

This is supplied to for the user, but it should be noted that iosDrvRemove() may cause unpredictable results.
usbSpkrExit()

NAME
usbSpkrExit() – de-initializes USB speaker SEQ_DEV driver

SYNOPSIS
STATUS usbSpkrExit(void)

DESCRIPTION
This function un-initializes the speaker class driver. It also un-registers the callback function and deletes the mutex.

RETURNS
OK or ERROR, if not able to un-initialize the speaker driver

ERRNO
none

SEE ALSO
usrUsbAudioDemo

usbSpkrInit()

NAME
usbSpkrInit() – initializes USB speaker SEQ_DEV driver

SYNOPSIS
STATUS usbSpkrInit(void)

DESCRIPTION
none

RETURNS
ERROR or OK

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo
**usbTargControlPayloadRcv()**

**NAME**
`usbTargControlPayloadRcv()` – receives data on the default control pipe

**SYNOPSIS**
```c
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 Application Callback */
)
```

**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**
```c
STATUS usbTargControlResponseSend
(  USB_TARG_CHANNEL targChannel,  /* target channel */
  UINT16           bfrLen,       /* length of response 0 */
  pUINT8           pBfr          /* ptr to bfr */
)
```

**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.
**2 Routines**

**usbTargControlStatusSend( )**

**NAME**
usbTargControlStatusSend( ) – sends control transfer status to the host

**SYNOPSIS**
STATUS usbTargControlStatusSend
    (USB_TARG_CHANNEL targChannel  /* target channel */)

**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 */)

---

ERRNO
S_usbTargLib_GENERAL_FAULT
Fault occurred in upper layers.

S_usbTargLib_BAD_PARAM
Bad Parameter is passed.

**SEE ALSO**
usbTargDefaultPipe
DESCRIPTION
This function allows a caller to retrieve the current USB frame number for the bus to which
\text{TargChannel} is connected. Upon return, the current frame number is stored in \text{pFrameNo}.

RETURNS
\text{OK}, or \text{ERROR} if unable to retrieve USB frame number

ERRNO
\text{S_usbTargLib\_TCD\_FAULT}
Fault occurred in TCD

SEE ALSO
\text{usbTargDeviceControl}

\textbf{usbTargDeviceFeatureClear( )}

NAME
\text{usbTargDeviceFeatureClear( )} – clears a specific feature

SYNOPSIS
\begin{verbatim}
STATUS usbTargDeviceFeatureClear
  ( \text{USB\_TARG\_CHANNEL targChannel,} /* target channel */
    \text{UINT16 ufeatureSelector} /* feature to be cleared */
  )
\end{verbatim}

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

RETURNS
\text{OK} or \text{ERROR} if not able to clear the feature.

ERRNO
\text{S_usbTargLib\_TCD\_FAULT}
Fault occurred in TCD.

SEE ALSO
\text{usbTargDeviceControl}

\textbf{usbTargDeviceFeatureSet( )}

NAME
\text{usbTargDeviceFeatureSet( )} – sets or enable a specific feature

SYNOPSIS
\begin{verbatim}
STATUS usbTargDeviceFeatureSet
  ( \text{USB\_TARG\_CHANNEL targChannel,} /* target channel */
    \text{UINT16 ufeatureSelector,} /* feature to be set */
    \text{UINT8 uTestSelector} /* test selector value */
  )
\end{verbatim}

DESCRIPTION
This function is used to set or enable a device specific feature.
usbTargDisable()

NAME

usbTargDisable() – disables a target channel

SYNOPSIS

STATUS usbTargDisable
   (USB_TARG_CHANNEL targChannel /* target to disable */)

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

STATUS usbTargEnable
   (USB_TARG_CHANNEL targChannel /* target to enable */)

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.
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 occured in TCD.

S_usbTargLib_BAD_PARAM
Bad parameter is passed.

SEE ALSO usbTargInitExit

usbTargInitialize()  

NAME usbTargInitialize() – initializes the USB Target Library  

SYNOPSIS STATUS usbTargInitialize (void)  

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 VOID usbTargKbdCallbackInfo
                      (
                          struct usbTargCallbackTable ** ppCallbacks,  /* Callback table pointer */
                      */
**2 Routines**

`usbTargKbdInjectReport()`

```c
pVOID                          *pCallbackParam /* target app-specific parameter */
```

**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**
```c
STATUS usbTargKbdInjectReport
{
    pHID_KBD_BOOT_REPORT pReport, /* Boot Report to be injected */
    UINT16             reportLen   /* Length of the boot report */
}
```

**DESCRIPTION**
This function injects the boot report into the interrupt pipe. `pReport` is the pointer to the boot report to be injected. `reportErpCallback` 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

STATUS usbTargMgmtCallback

(  
    pVOID  pTargTcd,   /* pointer to TARG_TCD structure */  
    UINT16 mngmtCode, /* management event code */  
    pVOID  pContext    /* parameter of management event */  
)

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 occured in TCD

SEE ALSO

usbTargDeviceControl

usbTargMsCallbackInfo()

NAME

usbTargMsCallbackInfo() – returns usbTargPrnLib callback table

SYNOPSIS

VOID usbTargMsCallbackInfo

(  
    struct usbTargCallbackTable ** ppCallbacks,    /*USB_TARG_CALLBACK_TABLE */  
    pVOID                      * pCallbackParam  /* Callback Parameter */  
)

DESCRIPTION

This function returns the callback table pointer.

RETURNS

N/A

ERRNO

none

SEE ALSO

usbTargMsLib
usbTargMsInit()

NAME
usbTargMsInit() – initializes USB mass storage functionality driver

SYNOPSIS
STATUS usbTargMsInit(void)

DESCRIPTION
This function initializes the peripheral stack and attaches it with the included Target Controller with mass storage functionality.

RETURNS
OK or ERROR if not able to attach to the TCD

ERRNO
none

SEE ALSO
usrUsbTargMsInit

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 occurred 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
STATUS usbTargPipeDestroy
    (USB_TARG_PIPE pipeHandle /* pipe to be destroyed */)

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 occured in TCD.

SEE ALSO
usbTargPipeFunc

usbTargPipeStatusGet( )

NAME
usbTargPipeStatusGet( ) – returns the endpoint status

SYNOPSIS
STATUS usbTargPipeStatusGet
    (USB_TARG_PIPE pipeHandle, /* Handle to the pipe */
    pUINT8 pBuf /* Buffer to hold the pipe status */)

DESCRIPTION
This function is used to get the status of the pipe as 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 occured in TCD.

SEE ALSO
usbTargPipeFunc
usbTargPipeStatusSet()

NAME
usbTargPipeStatusSet() – sets pipe stalled/unstalled status

SYNOPSIS
STATUS usbTargPipeStatusSet
    (USB_TARG_PIPE pipeHandle, /* Handle to the pipe */
     UINT16 state       /* State of the pipe to be set */
    )

DESCRIPTION
If the target application detects an error while servicing a pipe, it may choose to stall the
endpoint(s) associated with that pipe.
This function allows the caller to set the state of a pipe as "stalled" or "un-stalled".

RETURNS
OK, or ERROR if unable to set indicated state

ERRNO
S_usbTargLib_TCD_FAULT
    Fault occured in TCD.

SEE ALSO
usbTargPipeFunc

usbTargPrnCallbackInfo()

NAME
usbTargPrnCallbackInfo() – returns usbTargPrnLib callback table

SYNOPSIS
VOID usbTargPrnCallbackInfo
    (pUSB_TARG_CALLBACK_TABLE *ppCallbacks, /* Pointer to callback */
    ppCallbacksTable, /* table */
    pVOID *pCallbackParam /* target app-specific */
    *pCallbackParam /* parameter */
    )

DESCRIPTION
This function is called by the initialization routine. It returns the information about the
callback table.

RETURNS
N/A

ERRNO
none

SEE ALSO
usbTargPrnLib
usbTargPrnDataInfo()  

NAME  
usbTargPrnDataInfo() – returns buffer status/info  

SYNOPSIS  
STATUS usbTargPrnDataInfo  
   (  
        pUINT8  * ppBfr,   /* Pointer to the buffer address */  
        pUINT16   pActLen  /* Actual length of the data */  
   )  

DESCRIPTION  
This function returns the status the bulk buffer which consist of the data sent by the printer.  
$pActLen$ 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  
STATUS usbTargPrnDataRestart (void)  

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**
```c
STATUS usbTargRbcBlockDevCreate (void)
```

**DESCRIPTION**
This routine creates an RBC BLK I/O device. The RAM driver will be used for the actual implementation.

**RETURNS**
Pointer to `BLK_DEV` structure

**ERRNO**
none.

**SEE ALSO**
usbTargRbcCmd

### usbTargRbcBlockDevGet()

**NAME**
usbTargRbcBlockDevGet() – return opaque pointer to the RBC BLK I/O DEV device

**SYNOPSIS**
```c
pVOID usbTargRbcBlockDevGet (void)
```

**DESCRIPTION**
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**

```
STATUS usbTargRbcBlockDevSet
    (pVOID *blkDev /* pointer to the BLK_DEV device */)
```

**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 ** pData, /* 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**
```
STATUS usbTargRbcCapacityRead
{
    UINT8  arg[10],   /* RBC command */
    UINT8  **pData,  /* point to capacity data */
    UINT32 *pSize     /* size of capacity */
}
```

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

**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**
```c
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

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
usbTargRbcPersistentReserveIn()

NAME
usbTargRbcPersistentReserveIn() – send reserve data to the host

SYNOPSIS
STATUS usbTargRbcPersistentReserveIn
{
    UINT8 arg[10], /* the RBC command */
    UINT8 ** ppData, /* location of reserve data on device */
    UINT32 * pSize /* size of reserve data */
}

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
STATUS usbTargRbcPersistentReserveOut
{
    UINT8 arg[10], /* the RBC command */
    UINT8 ** ppData, /* location of reserve data on device */
    UINT32 * pSize /* size of reserve data */
}

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 STATUS usbTargRbcPreventAllowRemoval

   { UINT8 arg[6] /* the RBC command */}

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

SEE ALSO usbTargRbcCmd
**usbTargRbcRelease()**

**NAME**

`usbTargRbcRelease()` – release a resource on the RBC device

**SYNOPSIS**

```c
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**

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

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

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  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 */)

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**  
`VOID usbTargSetupErpCallback(  
    pUSB_ERP pErp /* Pointer to ERP structure */  
)`

**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 varable is used.

S_usbTargLib_TCD_FAULT  
Fault occured in TCD.

S_usbTargLib_APP_FAULT  
Application Specific fault occured.

**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
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 */
    )

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

S_usbTargLib_OUT_OF_MEMORY
   Memory not present to allocate variables.

S_usbTargLib_TCD_FAULT
   Fault occurred in TCD.

S_usbTargLib_OUT_OF_RESOURCES
   Sufficient resources not available.

S_usbTargLib_BAD_PARAM
   Bad parameter is passed.

S_usbTargLib_APP_FAULT
   Application Specific Fault occurred.

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(). targChannel 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 occurred in TCD.

S_usbTargLib_BAD_PARAM
   Bad parameter is passed.

S_usbTargLib_APP_FAULT
   Application Specific Fault occurred.

SEE ALSO

usbTargInitExit
usbTargTransfer( )

NAME
usbTargTransfer( ) – to transfer data through a pipe

SYNOPSIS
STATUS usbTargTransfer
    (USB_TARG_PIPE pipeHandle, /* handle to the pipe */
     pUSB_ERP   pErp       /* ERP to be transfered */
    )

DESCRIPTION
This function is used to initiate an transfer on the pipe indicated by pipeHandle. 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 occurred in TCD.
S_usbTargLib_BAD_PARAM
    Bad parameter is passed.

SEE ALSO
usbTargPipeFunc

usbTargTransferAbort( )

NAME
usbTargTransferAbort() – cancels a previously submitted USB_ERP

SYNOPSIS
STATUS usbTargTransferAbort
    (USB_TARG_PIPE pipeHandle, /* pipe for transfer to abort */
     pUSB_ERP   pErp       /* ERP to be aborted */
    )

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**

```c
STATUS usbTcdIsp1582EvalExec
    (pVOID pTrb  /* TRB to be executed */)
```

**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**

```c
STATUS usbTcdNET2280Exec
    (pVOID pTrb  /* TRB to be executed */)
```

**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 PDIUSBD12 TCD

SYNOPSIS  
STATUS usbtcdPdiusbd12EvalExec  
(  
pVOID pTrb  /* TRB to be executed */  
)

DESCRIPTION  
This is the single entry point for the Philips PDIUSBD12 (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

usbttool()  

NAME  
usbttool() – Primary entry point for USB bus exerciser.

SYNOPSIS  
UINT16 usbttool (void)

DESCRIPTION  
usbttool presents the user with a command prompt. The user enters commands to invoke  
USB functions through the USBD and USB HCD.

RETURNS  
RET_OK for normal termination;  
RET_ERROR for program failure.

ERRNO  
none

SEE ALSO  
usrUsbttool
**usbTransferTime()**

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

**SYNOPSIS**
```c
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 amount of 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() – uninitialise the USB UHCD Host Controller Driver.

**SYNOPSIS**
```c
USBHST_STATUS usbUhcdExit (void)
```

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

**RETURNS**
`USBHST_SUCCESS`, `USBHST_FALIURE` if the UHCD Host Controller uninitialization fails

**ERRNO**
None.

**SEE ALSO**
usbUhcdInitialization
usbUhcdInit()

NAME  usbUhcdInit() – initialise the USB UHCD Host Controller Driver.

SYNOPSIS  USBHST_STATUS usbUhcdInit (void)

DESCRIPTION  This routine initializes the UHCI Host Controller Driver and can be called from either the target initialization code (bootup) or during runtime. The usbd and hub interfaces should be initialized before this routine is called.

RETURNS  USBHST_FAILURE - if the UHCD Host Controller initialization fails

ERRNO  None.

SEE ALSO  usbUhcdInitialization

usbdAddressGet()

NAME  usbdAddressGet() – gets the USB address for a given device

SYNOPSIS  STATUS usbdAddressGet
{
    USBD_CLIENT_HANDLE clientHandle,  /* Client handle */
    USBD_NODE_ID nodeId,            /* Node Id of device/hub */
    pUINT16 pDeviceAddress     /* Currently assigned device address */
}

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

RETURNS  OK, or ERROR

ERRNO  none

SEE ALSO  usbTransUnitMisc
usbdAddressSet( )

NAME
usbdAddressSet( ) – sets the USB address for a given device

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

DESCRIPTION
This function 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 0..127. The deviceAddress must also
be unique within the scope of each USB host controller.

The USBD manages USB device addresses automatically, and this function 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
usbdTransUnitMisc

usbdBusCntGet( )

NAME
usbdBusCntGet( ) – obtain the number of USB's attached to the host

SYNOPSIS
UINT16 usbdBusCntGet (void)

DESCRIPTION
This function is called by the Integration testing tool to obtain the number of USB's attached
to the host.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
usbdMisc
usbdBusCountGet()

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

SYNOPSIS

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

DESCRIPTION
This function 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; and clients planning to
enumerate USB devices using the Bus Enumeration Functions need to know the total
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 Functions 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 (e.g., suspend/resume)

SYNOPSIS

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

DESCRIPTION
This function allows a client to set the state of the bus to which the specified *nodeId* is
attached. The desired *busState* is specified as USBD_BUS_xxxx.
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.

RETURNS OK, or ERROR if unable to set specified bus state

ERRNO N/A

SEE ALSO usbTransUnitInit

### usbdClientRegister( )

**NAME**
usbdClientRegister( ) – Registers a new client with the USBD

**SYNOPSIS**

```c
STATUS usbdClientRegister
    ( pCHAR pClientName,  /* Client name */
      pUSBD_CLIENT_HANDLE pClientHandle  /* Client hdl returned by USBD */
    )
```

**DESCRIPTION**
This routine invokes the USBD function to register a new client. `pClientName` should point to a string of not more than `USBD_NAME_LEN` characters (excluding terminating NULL) which can be used to uniquely identify the client. If successful, upon return the `pClientHandle` will be filled with a newly assigned `USBD_CLIENT_HANDLE`.

**RETURNS**
OK, or ERROR if unable to register new client.

**ERRNO**
N/A

**SEE ALSO**
usbTransUnitInit

### usbdClientUnregister( )

**NAME**
usbdClientUnregister( ) – Unregisters a USB client

**SYNOPSIS**

```c
STATUS usbdClientUnregister
    ( USBD_CLIENT_HANDLE clientHandle  /* Client handle */
    )
```
DESCRIPTION
A client invokes this function to release a previously assigned USBD_CLIENT_HANDLE. The USBD will release all resources allocated to the client, aborting any outstanding URBs which may exist for the client.

Once this function has been called with a given clientHandle, the client must not attempt to reuse the indicated clientHandle.

RETURNS
OK, or ERROR if unable to unregister client.

ERRNO
N/A

SEE ALSO
usbTransUnitInit

usbdConfigurationGet()

NAME
usbdConfigurationGet() – gets USB configuration for a device

SYNOPSIS
STATUS usbdConfigurationGet
{
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID nodeId,           /* Node Id of device/hub */
    pUINT16            pConfiguration /* bfr to receive config value */
}

DESCRIPTION
This function returns the currently selected configuration for the device or hub indicated by nodeId. The current configuration value is returned in the low byte of pConfiguration. The high byte is currently reserved and will be 0.

RETURNS
OK, or ERROR if unable to get configuration.

ERRNO
none

SEE ALSO
usbTransUnitStd
usbdConfigurationSet()

NAME
usbdConfigurationSet() – sets USB configuration for a device

SYNOPSIS
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 */
}

DESCRIPTION
This function sets the current configuration for the device identified by nodeId. The client should pass the desired configuration value in the low byte of configuration. The high byte is currently reserved and should be 0.

The client must also pass the maximum current which will be used by this configuration in maxPower.

RETURNS
OK, or ERROR if unable to set configuration.

ERRNO
none

SEE ALSO
usbdCurrentFrameGet

usbdCurrentFrameGet()

NAME
usbdCurrentFrameGet() – returns the current frame number for a USB

SYNOPSIS
STATUS usbdCurrentFrameGet
{
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID nodeId, /* Node Id of device/hub */
    UINT32* pFrameNo, /* bfr to receive current frame no. */
    UINT32* pFrameWindow /* bfr to receive frame window */
}

DESCRIPTION
It is sometimes necessary for clients to retrieve the current USB frame number for a specified host controller. This function allows a client to retrieve the current USB frame number for the host controller to which nodeId is connected. Upon return, the current frame number is stored in pFrameNo.

If pFrameWindow 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
2 Routines

usbdDescriptorGet()

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 1024 or 2048 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 \textit{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 \textit{nodeId} 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 \texttt{ERROR} if unable to retrieve current frame number.

ERRNO
none

SEE ALSO
usbTransUnitMisc

usbdDescriptorGet()

NAME
usbdDescriptorGet() – retrieves a USB descriptor

SYNOPSIS

```c
STATUS usbdDescriptorGet
    (
        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 \textit{nodeId}. \textit{requestType} is defined as documented for the \texttt{usbdFeatureClear()} function. \textit{descriptorType} specifies the type of the descriptor to be retrieved and must be one of the following values:

\textbf{USB\_DESCR\_DEVICE}

- Specifies the DEVICE descriptor.

\textbf{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, strings descriptors are returned in UNICODE format and the languageId should be the "sixteen-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 0.

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 descriptor.

ERRNO none

SEE ALSO usbTransUnitStd

usbdDescriptorSet( )

NAME usbdDescriptorSet() – sets a USB descriptor

SYNOPSIS STATUS usbdDescriptorSet
  {
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_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 */
  }

242
**DESCRIPTION**

A client uses this function to set a descriptor on the USB device identified by `nodeId`. The parameters `requestType`, `descriptorType`, `descriptorIndex`, and `languageId` are the same as those described for the `usbdDescriptorGet()` function. `pBfr` is a pointer to a buffer of length `bfrLen` 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`

---

**usbdDynamicAttachRegister()**

**NAME**

`usbdDynamicAttachRegister()` – Registers client for dynamic attach notification

**SYNOPSIS**

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

**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 the vendor specific devices) is attached 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 client’s convenience, `usbdLib.h` automatically includes `usb.h`, which defines a number of USB device classes as `USB_CLASS_xxxx` and `USB_SUBCLASS_xxxx`. A value of `USBD_NOTIFY_ALL` in any/all 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 a vendor specific class driver or FALSE, if it is a USB class specific class driver.

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

```c
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 an already-attached 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**: USBD is notifying the client that *nodeId* is a device which is now attached to the system.
- **USBD_DYNA_REMOVE**: USBD is notifying the client that *nodeId* 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 *nodeId*, such as data transfer requests, then the USBD will fail those outstanding requests prior to 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.

A client may re-use a single **attachCallback** for multiple notification registrations. As a convenience to the **attachCallback** routine, the USBD also passes the *deviceClass*, *deviceSubClass*, and *deviceProtocol* of the attached/removed *nodeId* each time it calls the **attachCallback**.

Finally, clients need to be aware that not all USB devices report their class information at the "device" level. Rather, some devices report class types on an interface-by-interface basis. When the device reports class information at the device level, then the USBD passes a configuration value of zero to the attach callback and calls the callback only a single time for each device. When the device reports class information at the interface level, then the Translation Unit invokes the attach callback once for each interface which matches the client's *deviceClass*/*deviceSubClass*/*deviceProtocol* specification. In this case, the USBD also
passes the corresponding configuration and interface numbers in configuration and interface each time it invokes the callback.

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

**ERRNO**
N/A

**SEE ALSO**
usbTransUnitInit

---

**usbdDynamicAttachUnRegister()**

**NAME**
usbdDynamicAttachUnRegister() – Unregisters client for attach notification

**SYNOPSIS**

```
STATUS usbdDynamicAttachUnRegister
    (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 for the attachment and removal of devices within the specified class. deviceClass, deviceSubClass, deviceProtocol, and attachCallback are defined as for the usbdDynamicAttachRegister() function and must match exactly the parameters passed in an earlier call to usbdDynamicAttachRegister.

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

**ERRNO**
N/A

**SEE ALSO**
usbTransUnitInit

---

**usbdExit()**

**NAME**
usbdExit() – exits USB2.0

**SYNOPSIS**

```
USBHST_STATUS usbdExit(void)
```
DESCRIPTION

This routine frees up memory allocated for the USBD2.0 layer and should only be called when bringing the USB2.0 stack down.

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:

USB_RT_DEVICE

Device

USB_RT_INTERFACE

Interface

USB_RT_ENDPOINT

Endpoint

requestType also specifies if the request is standard, class-specific, etc., as:

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 device's feature selector 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

### usbdFeatureSet()

**NAME**
usbdFeatureSet() – sets a USB feature

**SYNOPSIS**

```
STATUS usbdFeatureSet
(  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 feature as defined for the `usbdFeatureClear()` function.

The client must pass the device's feature selector 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

### usbdFrameWindowGet()

**NAME**
usbdFrameWindowGet() – obtain Frame Window and current frame number from HCD
SYNOPSIS

```c
USBHST_STATUS usbdFrameWindowGet
(  UINT32  hDevice,
      UINT16 pFrameWindow
)
```

DESCRIPTION

This function is called by the Integration testing tool to obtain the Frame Window and current frame number from HCD.

RETURNS

USBHST_SUCCESS, USBHST_INVALID_PARAMETER, USBHST_FAILURE if unsuccessful

ERRNO

N/A

SEE ALSO

usbdMisc

usbdHcdAttach()

NAME

`usbdHcdAttach()` – attaches an HCD to the USBD

SYNOPSIS

```c
STATUS usbdHcdAttach
(  HCD_EXEC_FUNC   hcdExecFunc,   /* Ptr to HCDís primary entry point */
     void *          hcdPciCfgHdr,  /* HCD-specific parameter */
     pGENERIC_HANDLE pAttachToken   /* Token to identify HCD in future */
)
```

DESCRIPTION

The `hcdExecFunc` passed by the caller must point to an HCDís primary entry point as defined below:

```c
typedef UINT16 (*HCD_EXEC_FUNC) (pHRB_HEADER pHrb);
```

RETURNS

OK

ERRNO

none

SEE ALSO

usbdTransUnitMisc

usbdHcdDetach()

NAME

`usbdHcdDetach()` – Detaches an HCD from the USBD

SYNOPSIS

```c
STATUS usbdHcdDetach
(  )
```
DESCRIPTION

The attachToken must be the attach token originally returned by usbdHcdAttach() when it first attached to the HCD.

RETURNS

OK

ERRNO

none

SEE ALSO

usbTransUnitMisc

---

**usbdHubPortCountGet()**

NAME

usbdHubPortCountGet() – returns number of ports connected to a hub

SYNOPSIS

```c
STATUS usbdHubPortCountGet(
    USBD_CLIENT_HANDLE clientHandle,  /* Client handle */
    USBD_NODE_ID hubId,         /* Node Id for desired hub */
    pUINT16 pPortCount     /* bfr to receive port count */
)
```

DESCRIPTION

usbdHubPortCountGet() provides clients with a convenient mechanism 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 Functions describe in a following section.

*hubId* must be the Node Id for the desired USB hub. An error will be returned if *hubId* does not refer to a hub. *pPortCount* must point to a UINT16 variable in which the total number of ports on the specified hub will be stored.

RETURNS

OK, or ERROR if unable to get hub port count.

ERRNO

none

SEE ALSO

usbTransUnitMisc
usbdInit( )

NAME    usbdInit( ) – initializes USBD2.0

SYNOPSIS  USBHST_STATUS usbdInit (void)

DESCRIPTION  This routine initializes the global variables for the USBD2.0 layer. It should be called before any hub, hcd, or class driver initialization code.

RETURNS  USBHST_SUCCESS, USBHST_FAILURE if event’s could not be created

ERRNO  None

SEE ALSO  usbd

usbdInitialize( )

NAME    usbdInitialize( ) – Initialize the USBD

SYNOPSIS  STATUS usbdInitialize (void)

DESCRIPTION  usbdInitialize( ) must be called at least once prior to calling other USBD functions. usbdInitialize( ) prepares the USBD and Translation unit to process URBs. Calls to usbdInitialize( ) may be nested, allowing multiple USBD clients to be written independently.

RETURNS  OK, or ERROR if initialization failed.

ERRNO  N/A

SEE ALSO  usbTransUnitInit

usbdInterfaceGet( )

NAME    usbdInterfaceGet( ) – retrieves a device's current interface

SYNOPSIS  STATUS usbdInterfaceGet
2 Routines

usbdInterfaceSet()

{ 
    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 */
}

DESCRIPTION
This function allows a client to query the current alternate setting for a given device's interface. nodeId and interfaceIndex specify the device and interface to be queried, respectively. pAlternateSetting points to a UINT16 variable in which the alternate setting will be stored upon return.

RETURNS
OK, or ERROR if unable to get interface.

ERRNO
none

SEE ALSO
usbTransUnitStd

usbdInterfaceSet( )

NAME
usbdInterfaceSet( ) – sets a device's current interface

SYNOPSIS
STATUS usbdInterfaceSet
{ 
    USBD_CLIENT_HANDLE clientHandle,      /* Client handle */
    USBD_NODE_ID nodeId,            /* Node Id of device/hub */
    UINT16 interfaceIndex,    /* Index of interface */
    UINT16 alternateSetting  /* Alternate setting */
}

DESCRIPTION
This function allows a client to select an alternate setting for a given device's interface. nodeId and interfaceIndex specify the device and interface to be modified, respectively. alternateSetting specifies the new alternate setting.

RETURNS
OK, or ERROR if unable to set interface.

ERRNO
none

SEE ALSO
usbTransUnitStd
usbdMngmtCallbackSet( )

NAME
usbdMngmtCallbackSet( ) – sets management callback for a client

SYNOPSIS
STATUS usbdMngmtCallbackSet
  (  
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */  
    USBD_MNGMT_CALLBACK mngmtCallback, /* management callback */  
    pVOID mngmtCallbackParam  /* client-defined parameter */  
  )

DESCRIPTION
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 usbdBusStateSet( ) - and a USB device drives RESUME signalling, that event can be reported to a client through its management callback.

clientHandle is a client's registered handled with the USBD. mngmtCallback is the management callback routine of type USBD_MNGMT_CALLBACK which will be invoked by the USBD when management events are detected. mngmtCallbackParam is a client-defined parameter which will be passed to the mngmtCallback each time it is invoked. Passing a mngmtCallback of NULL cancels management event callbacks.

When the mngmtCallback is invoked, the USBD will also pass 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 at most one management callback per USBD_CLIENT_HANDLE.

RETURNS
OK, or ERROR if unable to register management callback

ERRNO
N/A

SEE ALSO
usbTransUnitInit

usbdNodeIdGet( )

NAME
usbdNodeIdGet( ) – gets the id of the node connected to a hub port

SYNOPSIS
STATUS usbdNodeIdGet
  (  
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */  
    USBD_NODE_ID hubId, /* Node Id for desired hub */  
  )
Clients use this function to retrieve the Node Id for devices attached to each of a hub’s ports. `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 device (non-hub) is attached to the specified port.

If the node type is returned as `USB_NODETYPE_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 that hub or device’s nodeId 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**
```c
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 function 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:

```c
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. Currently 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 0.

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.

It is anticipated that this structure may grow over time. 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

usbdPipeCreate()

NAME

usbdPipeCreate() – Creates a USB pipe for subsequent transfers

SYNOPSIS

STATUS usbdPipeCreate 
( 
    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 */

    */
);
This function establishes a pipe which can subsequently 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, respectively, 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 USBD 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 USBD Configuration Functions and 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 0. 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. So, 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 0.

If the USBD 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 USBD Transfer Functions. If there is insufficient bus bandwidth available to create the pipe (as might happen for an isochronous or interrupt pipe), then the USBD 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
**usbdPipeDestroy()**

**NAME**
usbdPipeDestroy() – Destroys a USB data transfer pipe

**SYNOPSIS**

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

**DESCRIPTION**
This function destroys a pipe previously created by calling usbdPipeCreate(). The caller must pass the `pipeHandle` originally returned by usbdPipeCreate().

**RETURNS**
OK, or ERROR if unable to destroy pipe.

**ERRNO**
N/A

**SEE ALSO**
usbdTransUnitData

---

**usbdRootNodeIDGet()**

**NAME**
usbdRootNodeIDGet() – obtain node id of the root hub on a Host Controller

**SYNOPSIS**

```c
USBHST_STATUS usbdRootNodeIDGet
    ( UINT16  uBusIndex,
      PUINT32 phRootHub
    )
```

**DESCRIPTION**
This function is called by the Integration testing tool to obtain the node id of the root hub on a Host Controller.

**RETURNS**
USBHST_SUCCESS, USBHST_INVALID_PARAMETER, USBHST_FAILURE if unsuccessful

**ERRNO**
N/A

**SEE ALSO**
usbdMisc
**usbdRootNodeIdGet( )**

**NAME**

usbdRootNodeIdGet( ) – returns root node for a specific USB

**SYNOPSIS**

```c
STATUS usbdRootNodeIdGet
{
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    UINT16             busIndex,      /* Bus index */
    pUSBD_NODE_ID      pRootId        /* bfr to receive Root Id */
}
```

**DESCRIPTION**

This function returns the Node Id for the root hub for the specified USB host controller. `busIndex` is the index of the desired USB host controller. The first host controller is index 0 and the last host controller's index is the total number of USB host controllers - as returned by `usbdBusCountGet()` minus 1. `<pRootId>` must point to a `USBD_NODE_ID` variable in which the Node Id of the root hub will be stored.

**RETURNS**

OK, or ERROR if unable to get root node ID.

**ERRNO**

none

**SEE ALSO**

`usbTransUnitMisc`

---

**usbdShUTDOWN()**

**NAME**

usbdShUTDOWN( ) – Shuts down the USBD

**SYNOPSIS**

```c
STATUS usbdShUTDOWN (void)
```

**DESCRIPTION**

`usbdShUTDOWN()` should be called once for every successful call to `usbdInitialize()`. This function frees memory and other resources used by the USBD and Translation Unit.

**RETURNS**

OK, or ERROR if shutdown failed.

**ERRNO**

N/A

**SEE ALSO**

`usbTransUnitInit`
usbdStatisticsGet()

NAME  
usbdStatisticsGet() – Retrieves USBD operating statistics

SYNOPSIS  
STATUS usbdStatisticsGet
{
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID nodeId,        /* Node Id of node on desired USB */
    pUSBD_STATS pStatistics,   /* Ptr to structure to receive stats */
    UINT16 statLen        /* LEN of bfr provided by caller */
}

DESCRIPTION  
This function returns operating statistics for the USB to which the specified nodeId is connected.

The USBD copies the current operating statistics into the pStatistics structure provided by the caller. This structure is defined as:

typedef struct usbd_stats
{
    UINT16 totalTransfersIn;
    UINT16 totalTransfersOut;
    UINT16 totalReceiveErrors;
    UINT16 totalTransmitErrors;
} USBD_STATS, *pUSBD_STATS;

It is anticipated that this structure may grow over time. To provide backwards compatibility, the client must pass the total size of the USBD_STATS structure it has allocated in statLen. The USBD will copy fields into this structure only up to the statLen indicated by the caller.

RETURNS  OK

ERRNO  N/A

SEE ALSO  usbTransUnitMisc

usbdStatusGet()

NAME  
usbdStatusGet() – Retrieves USB status from a device/interface/etc.

SYNOPSIS  
STATUS usbdStatusGet
{
    USBD_CLIENT_HANDLE clientHandle, /* Client handle */
    USBD_NODE_ID nodeId,        /* Node Id of device/hub */
    UINT16 requestType,       /* Selects device/interface/endpoint */
}
Wind River USB for VxWorks 6  
API Reference, 2.3

DESCRIPTION

This function retrieves the current status from the device indicated by \textit{nodeId}. \textit{requestType} indicates the nature of the desired status as documented for the \texttt{usbdFeatureClear()} function.

The status word is returned in \textit{pBfr}. The meaning of the status varies depending on whether it was queried from the device, an interface, or an endpoint, class-specific function, etc. as described in the USB Specification.

RETURNS

\texttt{OK}, or \texttt{ERROR} if unable to get status.

ERRNO

none

SEE ALSO

\texttt{usbTransUnitStd}

\textbf{usbdSynchFrameGet()}\texttt{( )} 

\textbf{NAME}

\texttt{usbdSynchFrameGet()} – Returns a device's isochronous synch. frame

\textbf{SYNOPSIS}

\begin{verbatim}
STATUS usbdSynchFrameGet(
    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 */
)
\end{verbatim}

\textbf{DESCRIPTION}

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

\texttt{nodeId} specifies the node to query and \texttt{endpoint} specifies the endpoint on that device. Upon return the device's frame number for the specified endpoint is returned in \texttt{pFrameNo}.

RETURNS

\texttt{OK}, or \texttt{ERROR} if unable to retrieve synch. frame.

ERRNO

none

SEE ALSO

\texttt{usbTransUnitStd}
usbdTransfer( )

NAME

usbdTransfer() – Initiates a transfer on a USB pipe

SYNOPSIS

STATUS usbdTransfer
{
    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 function to initiate a transfer on the 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 prior to invoking usbdTransfer().

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 usbdLink;  // used by USBD
    pVOID usbdPtr;
    LINK hcdLink;  // used by HCD
    pVOID hcdPtr;
    pVOID userPtr;
    UINT16 irpLen;
    int result;    // returned by USBD/HCD
    IRP_CALLBACK usbdCallback;  // used by USBD
    IRP_CALLBACK userCallback;
    UINT16 dataToggle;   // filled in by USBD
    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 default structure contains a single bfrList element, but clients may allocate a longer structure to accommodate a larger number of bfrList elements.

flags define 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 error

USB_Flag_ISO_ASP
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 or may not indicate an error condition depending on the design of the device. For many devices, the underrun is completely normal and indicates the end of data stream from the device. For other devices, the underrun indicates a transfer failure. By default, the USBD and underlying USB HCD (Host Controller Driver) treat underrun as the end-of-data indicator and do not declare an error. If the USB_Flag_Short_Fail flag is set, then the USBD/HCD will instead treat underrun as an error condition.

For isochronous transfers the USB_Flag_ISO_ASP specifies that the isochronous transfer should begin as soon as possible. If USB_Flag_ISO_ASP is not specified, then startFrame must specify the starting frame number for the transfer. The usbdCurrentFrameGet() function 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-0, the dataBlockSize defines the granularity of isochronous data being sent. When the underlying Host Controller Driver (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 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 usbdTransfer() function returns as soon as the IRP has been successfully enqueued. If there
is a failure in delivering the IRP to the HCD, then `usbdTransfer()` returns an error. The actual result of the IRP should be checked after the `userCallback` routine has been invoked.

**RETURNS**

OK, or ERROR if unable to submit IRP for transfer.

**ERRNO**

N/A

**SEE ALSO**

`usbTransUnitData`

---

**usbdTransferAbort( )**

**NAME**

`usbdTransferAbort()` – Abort a transfer

**SYNOPSIS**

```c
STATUS usbdTransferAbort
(  USBD_CLIENT_HANDLE clientHandle,  /* Client handle */
  USBD_PIPE_HANDLE   pipeHandle,    /* Pipe handle */
  pUSB_IRP           pIrp           /* ptr to I/O to abort */
)
```

**DESCRIPTION**

This function aborts an IRP which was previously submitted through a call to `usbdTransfer()`.  

**RETURNS**

OK, or ERROR if unable to abort transfer.

**ERRNO**

N/A

**SEE ALSO**

`usbTransUnitData`

---

**usbdTranslationInit( )**

**NAME**

`usbdTranslationInit()` – initialize the USBD 1.1 Translation Layer

**SYNOPSIS**

```c
STATUS usbdTranslationInit(void)
```

**DESCRIPTION**

This function initializes the USBD 1.1 backwards compatibility layer that allows class drivers written with for the USB1.1 Host Stack to operate with the USB2.0 Host Stack.

**RETURNS**

OK if successful or ERROR if failure
ERRNO

none

SEE ALSO

usrUsbInit

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 functions described elsewhere. This function 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 function are always directed to the control pipe of the device specified by `nodeId`. This function 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 actual 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 USBD version information

**SYNOPSIS**

```c
STATUS usbdVersionGet
    (pUINT16 pVersion, /* UINT16 bfr to receive version */
     pCHAR   pMfg       /* bfr to receive USBD mfg string */)
```

**DESCRIPTION**

This function returns the USBD version. If `pVersion` is not NULL, the USBD returns its version in BCD in `pVersion`. For example, version "1.02" would be coded as 01h in the high byte and 02h in the low byte.

If `pMfg` is not NULL it must point to a buffer of at least USBD_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).

**RETURNS**

OK, or ERROR

**ERRNO**

none

**SEE ALSO**

usbtuDataUrbCompleteCallback()

---

**usbtuDataUrbCompleteCallback()**

**NAME**
usbtuDataUrbCompleteCallback() – Callback called on URB completeion

**SYNOPSIS**

```c
USBHST_STATUS usbtuDataUrbCompleteCallback
    (pUSBHST_URB urbPtr /* URB pointer */)
```

**DESCRIPTION**

This function is called from interrupt context by the USBD on a URB completion.

**RETURNS**

USBHST_SUCCESS, or USBHST_FAILURE on failure

**ERRNO**

N/A

**SEE ALSO**

usbTransUnitData
usbtuDataVendorSpecificCallback()

NAME  usbtuDataVendorSpecificCallback() – Callback called on Vendor Specific Request

SYNOPSIS  USBHST_STATUS usbtuDataVendorSpecificCallback
             (pUSBHST_URB urbPtr /* URB pointer */)

DESCRIPTION  completion

This function is called from interrupt context by the USBD on a Vendor Specific request completion.

RETURNS  USBHST_SUCCESS

ERRNO  N/A

SEE ALSO  usbTransUnitData

usbtuInitClientIrpCompleteThreadFn()

NAME  usbtuInitClientIrpCompleteThreadFn() – Client Thread Function

SYNOPSIS  VOID usbtuInitClientIrpCompleteThreadFn
             (pVOID driverParam)

DESCRIPTION  This function is executed by a client Thread. The thread waits on the message queue created for the client. The message is of the type USBTU_CLIENTMSG. Based on the USBTU_EVENTCODE in the message it performs the action.

RETURNS  N/A

ERRNO  N/A

SEE ALSO  usbTransUnitInit
usbtuInitClientThreadFn( )

NAME  
usbtuInitClientThreadFn() – Client Thread Function

SYNOPSIS  
VOID usbtuInitClientThreadFn(
    pVOID driverParam
)

DESCRIPTION  
This function is executed by a client Thread. The thread waits on the message queue created for the client. The message is of the type USBTU_CLIENTMSG. Based on the USBTU_EVENTCODE in the message it performs the action.

RETURNS  
N/A

ERRNO  
N/A

SEE ALSO  
usbtuInitDeviceAdd( )

usbtuInitDeviceAdd( )

NAME  
usbtuInitDeviceAdd() – Device Attach Callback

SYNOPSIS  
USBHST_STATUS usbtuInitDeviceAdd(
    UINT32 hDevice,
    UINT8  interfaceNumber,
    UINT8  speed,
    void** ppDriverData
)

DESCRIPTION  
This function is called from interrupt context by USBD on a device attach.

RETURNS  
USBHST_SUCCESS, or USBHST_FAILURE on failure

ERRNO  
N/A

SEE ALSO  
usbtuInitClientThreadFn()
### usbtuInitDeviceRemove()

**NAME**
usbtuInitDeviceRemove() – Device Detach Callback

**SYNOPSIS**
VOID usbtuInitDeviceRemove
    (UINT32 hDevice,
     PVOID  pDriverData
    )

**DESCRIPTION**
This function is called from interrupt context by USBD on a device detach.

**RETURNS**
N/A

**ERRNO**
N/A

**SEE ALSO**
usbTransUnitInit

### usbtuInitDeviceResume()

**NAME**
usbtuInitDeviceResume() – Device Resume Callback

**SYNOPSIS**
VOID usbtuInitDeviceResume
    (UINT32 hDevice,
     PVOID  pSuspendData
    )

**DESCRIPTION**
This function is called from interrupt context by USBD on a device resume.

**RETURNS**
N/A

**ERRNO**
N/A

**SEE ALSO**
usbTransUnitInit
### usbtuInitDeviceSuspend()

**NAME**  
usbtuInitDeviceSuspend() – Device Suspend Callback

**SYNOPSIS**  
VOID usbtuInitDeviceSuspend  

```c
  (  
    UINT32 hDevice,  
    PVOID ppSuspendData  
  )
```

**DESCRIPTION**  
This function is called from interrupt context by USBD on a device suspend.

**RETURNS**  
N/A

**ERRNO**  
N/A

**SEE ALSO**  
usbTransUnitInit

### usbtuInitThreadFn()

**NAME**  
usbtuInitThreadFn() – Translation Unit Thread Function

**SYNOPSIS**  
VOID usbtuInitThreadFn

```c
  (  
    pVOID param  /* User Parameter */  
  )
```

**DESCRIPTION**  
This function is executed by the Translation Unit Thread. The thread waits on the message queue created for the Translation Unit The message is of the type USBTU_TUMSG. Based on the USBTU_EVENTCODE in the message it performs appropriate actions

**RETURNS**  
N/A

**ERRNO**  
N/A

**SEE ALSO**  
usbTransUnitInit
usrUsbAudioDemo()

NAME
usrUsbAudioDemo() – Entry point to USB audio demo

SYNOPSIS
STATUS usrUsbAudioDemo (void)

DESCRIPTION
none

RETURNS
OK or ERROR

ERRNO
Not Available

SEE ALSO
usrUsbAudioDemo

usrUsbBulkDevDown()

NAME
usrUsbBulkDevDown() – un-initializes USB BULK Mass storage driver

SYNOPSIS
STATUS usrUsbBulkDevDown (void)

DESCRIPTION
This function un-initializes the BULK driver and un-registers a BULK drive with the USBD. It also un-mount the device if it is connected from the file system

RETURNS
OK or ERROR if error un-initializing the mass storage device

ERRNO
none

SEE ALSO
usrUsbBulkDevInit

usrUsbBulkDriveEmtpy()

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

SYNOPSIS
BOOL usrUsbBulkDriveEmpty
{
   char *driveName
}
**usrUsbBulkShow()**

**NAME**
usrUsbBulkShow() – shows routine for displaying all bulk devices.

**SYNOPSIS**
void usrUsbBulkShow (void)

**DESCRIPTION**
This function displays all the bulk devices

**RETURNS**
N/A

**ERRNO**
one

**SEE ALSO**
usrUsbBulkDevInit

**usrUsbCbiUfiDevDown()**

**NAME**
usrUsbCbiUfiDevDown() – un-initializes USB UFI Mass storage driver.

**SYNOPSIS**
STATUS usrUsbCbiUfiDevDown (void)

**DESCRIPTION**
This function un-initializes the USB UFI Mass storage driver

**RETURNS**
OK or ERROR if error in un-initializing

**ERRNO**
one

**SEE ALSO**
usrUsbCbiUfiDevInit
usrUsbCbiUfiDevInit()

NAME    usrUsbCbiUfiDevInit() – initializes USB UFI Mass storage driver.
SYNOPSIS STATUS usrUsbCbiUfiDevInit (void)
DESCRIPTION This function initializes the USB UFI Mass storage driver and then registers the client by calling the routine ufiClntRegister().
RETURNS OK or ERROR if error in initializing
ERRNO none
SEE ALSO usrUsbCbiUfiDevInit

usrUsbHcdEhciAttach()

NAME    usrUsbHcdEhciAttach() – attaches EHCI HCD to the USB Stack
SYNOPSIS STATUS usrUsbHcdEhciAttach (void)
DESCRIPTION This function searches for an EHCI type USB Host Controller. If it finds one, it attaches the HCD to the already initialized USB Stack
RETURNS OK if successful or ERROR if failure
ERRNO none
SEE ALSO usrUsbHcdEhciInit

usrUsbHcdEhciDetach()

NAME    usrUsbHcdEhciDetach() – detaches the EHCI HCD from the USB Stack
SYNOPSIS STATUS usrUsbHcdEhciDetach (void)
DESCRIPTION This function searches for an EHCI type USB Host Controller. If it finds one, it detaches the HCD from the stack.
usrUsbHcdOhciAttach()

NAME
usrUsbHcdOhciAttach() – attaches OHCI HCD to the USB Stack

SYNOPSIS
STATUS usrUsbHcdOhciAttach (void)

DESCRIPTION
This function searches for an OHCI type USB Host Controller. If it finds one, it attaches
the HCD to the already initialized USB Stack.

RETURNS
OK if successful or ERROR if failure

ERRNO
none

SEE ALSO
usrUsbHcdOhciInit

usrUsbHcdOhciDetach()

NAME
usrUsbHcdOhciDetach() – detaches the OHCI HCD from the USB Stack

SYNOPSIS
STATUS usrUsbHcdOhciDetach (void)

DESCRIPTION
This function searches for an OHCI type USB Host Controller. If it finds one, it detaches
the HCD from the USB Stack.

This function should not be called when a device is connected to the OHCI Host Controller.

RETURNS
OK if successful or ERROR if failure

ERRNO
none

SEE ALSO
usrUsbHcdOhciInit
### usrUsbHcdUhciAttach()

**NAME**  
usrUsbHcdUhciAttach() – attaches UHCI HCD to the USB Stack

**SYNOPSIS**  
STATUS usrUsbHcdUhciAttach (void)

**DESCRIPTION**  
This function searches for an UHCI type USB Host Controller. If it finds one, it attaches the HCD to the already initialized USB Stack.

**RETURNS**  
OK if successful or ERROR if failure

**ERRNO**  
none

**SEE ALSO**  
usrUsbHcdUhciInit

### usrUsbHcdUhciDetach()

**NAME**  
usrUsbHcdUhciDetach() – detaches UHCI HCD from the USB Stack

**SYNOPSIS**  
STATUS usrUsbHcdUhciDetach (void)

**DESCRIPTION**  
This function searches for an UHCI type USB Host Controller. If it finds one, it detaches the HCD from the USB Stack.

This function should not be called when a device is connected to the UHCI Host Controller.

**RETURNS**  
OK if successful or ERROR if failure

**ERRNO**  
none

**SEE ALSO**  
usrUsbHcdUhciInit

### usrUsbHubInit()

**NAME**  
usrUsbHubInit() – initialize the USB Hub Driver

**SYNOPSIS**  
STATUS usrUsbHubInit(void)
### usrUsbMseInit()

**NAME**
usrUsbMseInit() – initialize the USB mouse driver

**SYNOPSIS**
STATUS usrUsbMseInit (void)

**DESCRIPTION**
This function initializes the USB mouse driver and registers for attach callbacks.

**RETURNS**
OK or Error, if not able to initialize

**ERRNO**
none

**SEE ALSO**
usrUsbMseInit

### usrUsbPegasusEndDown()

**NAME**
usrUsbPegasusEndDown() – un-initialize the USB END Pegasus driver

**SYNOPSIS**
STATUS usrUsbPegasusEndDown (void)

**DESCRIPTION**
This function un-intializes the pegasus end driver. It also unregisters the callback function.

**RETURNS**
OK or ERROR

**ERRNO**
none

**SEE ALSO**
usrUsbPegasusEndInit
usrUsbPegasusEndInit()

NAME
usrUsbPegasusEndInit() – initialize the USB END Pegasus driver

SYNOPSIS
STATUS usrUsbPegasusEndInit (void)

DESCRIPTION
This function initializes the USB END Pegasus driver and spawns a task to manage device insertion / removals.

RETURNS
OK or ERROR

ERRNO
none

SEE ALSO
usrUsbPegasusEndInit

usrUsbPrnInit()

NAME
usrUsbPrnInit() – initialize the USB printer driver

SYNOPSIS
STATUS usrUsbPrnInit (void)

DESCRIPTION
This function initializes the USB printer driver and registers for attach callbacks.

RETURNS
OK or ERROR if not able to initialize

ERRNO
none

SEE ALSO
usrUsbPrnInit

usrUsbRawRead()

NAME
usrUsbRawRead() – raw read from the device

SYNOPSIS
STATUS usrUsbRawRead
{
    char * devName, /* Name of the Device */
    int blkNum, /* Block Number */
    char * pData /* Pointer to hold the data sent */
}
usrUsbRawWrite()

NAME
usrUsbRawWrite() – raw write to the device

SYNOPSIS

```c
STATUS usrUsbRawWrite
    (char * devName, /* Name of the device */
     int blkNum, /* Block Number */
     char * pData /* Data to be sent */
    )
```

DESCRIPTION
This function is to raw write to the device. The function is used for debugging purpose and send raw data over the bulk - out pipe.

RETURNS
OK on success, or ERROR

ERRNO
none

SEE ALSO
usrUsbBulkDevInit

usrUsbSpkrInit()

NAME
usrUsbSpkrInit() – initialize the USB speaker driver

SYNOPSIS

```c
STATUS usrUsbSpkrInit (void)
```

DESCRIPTION
This function initializes the USB speaker driver and registers for attach callbacks.

RETURNS
OK or ERROR if not able to initialize the speaker driver

ERRNO
none

SEE ALSO
usrUsbSpkrInit
### usrUsbTargKbdInit()

<table>
<thead>
<tr>
<th>NAME</th>
<th>usrUsbTargKbdInit() – initialization function for keyboard.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>void usrUsbTargKbdInit (void)</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This is the initialization function which associates the keyboard function driver to the TCD.</td>
</tr>
<tr>
<td>RETURNS</td>
<td>N/A</td>
</tr>
<tr>
<td>ERRNO</td>
<td>None.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>usrUsbTargKbdInit</td>
</tr>
</tbody>
</table>

### usrUsbTargPrnInit()

<table>
<thead>
<tr>
<th>NAME</th>
<th>usrUsbTargPrnInit() – Initialization function for printer.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>void usrUsbTargPrnInit (void)</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>This is the initialization function which associates the printer function driver to the TCD. It first initializes the peripheral stack by calling the routine Then it attaches the Printer Functionality with the Target Controller Driver</td>
</tr>
<tr>
<td>RETURNS</td>
<td>None.</td>
</tr>
<tr>
<td>ERRNO</td>
<td>None.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>usrUsbTargPrnInit</td>
</tr>
</tbody>
</table>
waitForSpeaker( )

NAME  waitForSpeaker( ) – waits for a speaker to be connected

SYNOPSIS  STATUS waitForSpeaker (void)

DESCRIPTION  none

RETURNS  OK if speaker connected, else ERROR

ERRNO  Not Available

SEE ALSO  usrUsbAudioDemo