QNX® Momentics® DDK

Universal Serial Bus (USB) Devices

For QNX® Neutrino® 6.3.0 or QNX® 4

© 2007, QNX Software Systems GmbH & Co. KG.
Miscellaneous and convenience functions

- `usbd_abort_pipe()` 18
- `usbd_alloc()` 19
- `usbd_alloc_urb()` 21
- `usbd_args_lookup()` 22
- `usbd_attach()` 23
- `usbd_close_pipe()` 25
- `usbd_configuration_descriptor()` 26
- `usbd_connect()` 28
- `usbd_descriptor()` 32
- `usbd_detach()` 34
- `usbd_device_descriptor()` 36
- `usbd_device_extra()` 38
- `usbd_device_lookup()` 39
- `usbd_disconnect()` 40
- `usbd_endpoint_descriptor()` 41
- `usbd_feature()` 43
- `usbd_free()` 45
- `usbd_free_urb()` 46
- `usbd_get_frame()` 47
- `usbd_hcd_ext_info()`, `usbd_hcd_info()` 48
- `usbd_hub_descriptor()` 50
- `usbd_interface_descriptor()` 52
- `usbd_io()` 54
- `usbd_languages_descriptor()` 56
- `usbd_mphys()` 58
- `usbd_open_pipe()` 59
- `usbd_parse_descriptors()` 61
- `usbd_pipe_device()` 63
- `usbd_pipe_endpoint()` 64
- `usbd_reset_device()` 65
- `usbd_reset_pipe()` 66
- `usbd_select_config()` 67
- `usbd_select_interface()` 68
- `usbd_setup_bulk()` 70
- `usbd_setup_control()` 72
- `usbd_setup_interrupt()` 74
- `usbd_setup_isochronous()` 76
- `usbd_setup_vendor()` 78
- `usbd_status()` 80
- `usbd_string()` 82
usbd_topology(), usbd_topology_ext()  84
usbd_urb_status()  86

Index  89
About the USB DDK
What you’ll find in this guide

The USB Driver Development Kit will help you write drivers for Universal Serial Bus devices.

Our USB API is designed to work with either QNX Neutrino or QNX 4. Exceptions will be noted where appropriate.

The following table may help you find information quickly:

<table>
<thead>
<tr>
<th>For information on:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>System requirements and other vital information</td>
<td>Before You Begin</td>
</tr>
<tr>
<td>How the OS supports USB</td>
<td>Overview</td>
</tr>
<tr>
<td>Command-line utilities</td>
<td>USB Utilities</td>
</tr>
<tr>
<td>USB driver interface calls</td>
<td>USB Library Reference</td>
</tr>
</tbody>
</table>

The USB DDK includes source code for several USB class drivers. Each class driver is contained in its own separate archive. Look under the `/ddk_working_dir/usb/src/hardware/devu/class` directory on your system.

Assumptions

We assume you’re familiar with the Universal Serial Bus (USB) Specification revision 2.0, especially the chapters on:

- Architectural Overview
- USB Data Flow Model
- USB Device Framework
- USB Host: Hardware and Software.

You’ll need a good understanding of the concepts in those chapters in order to write USB client device drivers.

For up-to-date information on USB developments, visit [www.usb.org](http://www.usb.org).

Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:
Please refer to the Managing Source Code chapter, and “QNX Source Package” in the Common Wizards Reference chapter of the IDE User’s Guide.

- To compile the DDK from the command line:
  
  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

  DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

  Historically, DDKs were placed in /usr/src/ddk\_VERSION directory, e.g. /usr/src/ddk-6.2.1. This method is no longer required, as each DDK archive is completely self-contained.

  The following example indicates how you create a directory and unzip the archive file:

  ```bash
  # cd ~
  # mkdir my_DDK
  # cd my_DDK
  # unzip /path_to_ddks/ddk\_device\_type.zip
  ```

  The top-level directory structure for the DDK looks like this:
You must run:

```
./setenv.sh
```
before running `make` or `make install`.

Additionally, on Windows hosts you’ll need to run the Bash shell (`bash.exe`) before you run the `./setenv.sh` command.

If you fail to run the `./setenv.sh` shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in `$QNX_TARGET`.

Each time you start a new shell, run the `./setenv.sh` command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell’s environment, not a sub-shell environment.
In **ksh** and **bash** shells, all shell scripts are executed in a sub-shell by default. Therefore, it’s important that you use the syntax

```bash
. <script>
```

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type `make` within this directory, you’ll generate all of the buildable entities within that DDK no matter where you move the directory.

All binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that’s built is installed outside of the DDK’s directory. The `make` files shipped with the DDKs copy the contents of the **prebuilt** directory into the **install** directory. The binaries are built from the source using include files and link libraries in the **install** directory.

### Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

<table>
<thead>
<tr>
<th>Reference</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code examples</td>
<td><code>if( stream == NULL )</code></td>
</tr>
<tr>
<td>Command options</td>
<td><code>-lR</code></td>
</tr>
<tr>
<td>Commands</td>
<td><code>make</code></td>
</tr>
<tr>
<td>Environment variables</td>
<td><code>PATH</code></td>
</tr>
<tr>
<td>File and pathnames</td>
<td><code>/dev/null</code></td>
</tr>
<tr>
<td>Function names</td>
<td><code>exit()</code></td>
</tr>
<tr>
<td>Keyboard chords</td>
<td>Ctrl-Alt-Delete</td>
</tr>
<tr>
<td>Keyboard input</td>
<td><code>something you type</code></td>
</tr>
<tr>
<td>Keyboard keys</td>
<td>Enter</td>
</tr>
<tr>
<td>Program output</td>
<td><code>login:</code></td>
</tr>
<tr>
<td>Programming constants</td>
<td><code>NULL</code></td>
</tr>
<tr>
<td>Programming data types</td>
<td><code>unsigned short</code></td>
</tr>
<tr>
<td>Programming literals</td>
<td><code>0xFF,&quot;message string&quot;</code></td>
</tr>
<tr>
<td>Variable names</td>
<td><code>stdin</code></td>
</tr>
</tbody>
</table>

*continued...*
We use an arrow (→) in directions for accessing menu items, like this:

You’ll find the Other... menu item under Perspective→Show View.

We use notes, cautions, and warnings to highlight important messages:

<table>
<thead>
<tr>
<th>Notes</th>
<th>CAUTION</th>
<th>WARNING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes point out something important or useful.</td>
<td>Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.</td>
<td>Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.</td>
</tr>
</tbody>
</table>

**Note to Windows users**

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

**Technical support**

To obtain technical support for any QNX product, visit the Support + Services area on our website (www.qnx.com). You’ll find a wide range of support options, including community forums.
Chapter 1
Before You Begin

In this chapter...

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>System requirements</td>
<td>3</td>
</tr>
<tr>
<td>USB devices supported</td>
<td>3</td>
</tr>
<tr>
<td>Known limitations</td>
<td>3</td>
</tr>
</tbody>
</table>
**System requirements**

This USB DDK is designed to work with both QNX Neutrino 6 and with QNX 4.

**For QNX Neutrino 6.3**

You’ll need the following:

- QNX Neutrino 6.3
- GNU GCC 2.95.2
- USB EHCI, OHCI or UHCI controller, version 1.1 and 2.0 compliant

**For QNX 4**

You’ll need the following:

- QNX 4.25, patch D or later
- Watcom 10.6, patch B or later
- USB EHCI, OHCI or UHCI controller, version 1.1 and 2.0 compliant

**USB devices supported**

<table>
<thead>
<tr>
<th>Type of device</th>
<th>Manufacturer</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Belkin</td>
<td>MediaBoard F8E211-USB</td>
</tr>
<tr>
<td>Mouse</td>
<td>Logitech</td>
<td>USB Wheel Mouse M-BB48</td>
</tr>
<tr>
<td></td>
<td>Micro Innovations</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>WingMan Gaming Mouse M-BC38</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Microsoft</td>
<td>IntelliMouse</td>
</tr>
<tr>
<td>Hub</td>
<td>ADS Technologies</td>
<td>4-port</td>
</tr>
<tr>
<td></td>
<td>Belkin</td>
<td>4-port</td>
</tr>
<tr>
<td>Printer</td>
<td>Canon</td>
<td>BJC-85</td>
</tr>
<tr>
<td></td>
<td>Epson</td>
<td>Stylus Color 740</td>
</tr>
<tr>
<td></td>
<td>HP</td>
<td>DeskJet 895Cse</td>
</tr>
</tbody>
</table>

**Known limitations**

**EHCI**

Isochronous and split isochronous transfers are unsupported at this time.

Retrieving ‘Other Speed Descriptor’ has not been implemented.
Photon and text mode

If you’re using Photon as well as text mode, you won’t be able to switch between them and use a USB keyboard once the USB stack has been started.

From a cold boot, you’ll be able to use a USB keyboard in text mode before the USB stack has been started. As soon as you start the USB stack, you can’t use a USB keyboard in text mode.

CAUTION:

Make sure that the command line for devi-hirun (or Input) includes the option to not reset the keyboard controller. For example:

```
devi-hirun kbd -R fd -d/dev/usbkbd0 &
```

Or with QNX 4:

```
Input kbd -R fd -d/dev/usbkbd0 &
```

If you don’t use the -R option, then the keyboard controller will be reset whenever you switch between Photon and text mode, and the machine may hang.
In this chapter...

- The USB stack and library 7
- How a class driver works 8
The USB stack and library

USB (Universal Serial Bus) is a hardware and protocol specification for interconnecting various devices to a host controller. We supply a USB stack that implements the USB protocol and allows user-written class drivers to communicate with USB devices.

We also supply a USB driver library (usbd_*) for class drivers to use in order to communicate with the USB stack. Note that a class driver can be considered a “client” of the USB stack.

The stack is implemented as a standalone process that registers the pathname of /dev/io-usb/io-usb (by default). Currently, the stack contains the hub class driver within it.

Host Controller Interface (HCI) types

The stack supports the three industry-standard HCI types:

- Open Host Controller Interface (OHCI)
- Universal Host Controller Interface (UHCI)
- Enhanced Host Controller Interface (EHCI)

We provide separate servers for each type (devu-ohci.so, devu-uhci.so, and devu-ehci.so). Note that USB devices don’t care whether a computer has an OHCI, UHCI, or an EHCI controller.

Data buffers

The client library provides functions to allocate data buffers in shared memory; the stack manages these data buffers and gives the client library access to them. This means that all data transfers must use the provided buffers.

As a result, a class driver must reside on the same physical node as the USB stack. The clients of the class driver, however, can be network-distributed. The advantage of this approach is that no additional memory copy occurs between the time that the data is received by the USB stack and the time that it’s delivered to the class driver (and vice versa).

USB enumerator

With the QNX Neutrino OS, the USB enumerator attaches to the USB stack and waits for device insertions. When a device insertion is detected, the enumerator looks in the configuration manager’s database to see which class driver it should start. It then starts the appropriate driver, which provides for that class of device. For example, a USB Ethernet class driver would register with io-net and bring the interface up.

For small, deeply embedded systems, the enumerator isn’t required. The class drivers can be started individually — they’ll wait around for their particular devices to be detected by the stack. At that point, they’ll provide the appropriate services for that
class of device, just as if they'd been started by the enumerator. When a device is removed, the enumerator will shut down the class driver.

For more information about device enumeration, see the Controlling How Neutrino Starts chapter of the Neutrino User’s Guide.

How a class driver works

A class driver typically performs the following operations:

1. Connect to the USB stack (usbd_connect()) and provide two callbacks: one for insertion and one for removal.

2. In the insertion callback:
   2a. Connect to the USB device (usbd_attach()).
   2b. Get descriptors (usbd_descriptor()).
   2c. Select the configuration (usbd_select_config()) and interface (usbd_select_interface()).
   2d. Set up communications pipes to the appropriate endpoint (usbd_open_pipe()).

3. In the removal callback, detach from the USB device (usbd_detach()).

4. Set up all data communications (e.g. reading and writing data, sending and receiving control information, etc.) via the usbd_setup_*() functions (usbd_setup_bulk(), usbd_setup_interrupt(), etc.).

5. Initiate data transfer using the usbd_io() function (with completion callbacks if required).

In this context, the term “pipe” is a USB-specific term that has nothing to do with standard POSIX “pipes” (as used, for example, in the command line ls | more). In USB terminology, a “pipe” is simply a handle; something that identifies a connection to an endpoint.
The USB Software Development Kit contains the following command-line utilities. For more information, see their entries in the Utilities Reference.

- **devu-ehci.so**: USB manager for Enhanced Host Controller Interface standard controllers. (USB 2.0)
- **devu-ohci.so**: USB manager for Open Host Controller Interface standard controllers. (USB 2.0)
- **devu-prn**: Class Driver for USB printers.
- **devu-uhci.so**: USB manager for Universal Host Controller Interface standard controllers. (USB 2.0)
- **io-usb**: USB server.
- **usb**: Display USB device configuration.
## In this chapter...

Functions arranged by category

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>usbd_abort_pipe()</code></td>
<td>18</td>
</tr>
<tr>
<td><code>usbd_alloc()</code></td>
<td>19</td>
</tr>
<tr>
<td><code>usbd_alloc_urb()</code></td>
<td>21</td>
</tr>
<tr>
<td><code>usbd_args_lookup()</code></td>
<td>22</td>
</tr>
<tr>
<td><code>usbd_attach()</code></td>
<td>23</td>
</tr>
<tr>
<td><code>usbd_close_pipe()</code></td>
<td>25</td>
</tr>
<tr>
<td><code>usbd_configuration_descriptor()</code></td>
<td>26</td>
</tr>
<tr>
<td><code>usbd_connect()</code></td>
<td>28</td>
</tr>
<tr>
<td><code>usbd_descriptor()</code></td>
<td>32</td>
</tr>
<tr>
<td><code>usbd_detach()</code></td>
<td>34</td>
</tr>
<tr>
<td><code>usbd_device_descriptor()</code></td>
<td>36</td>
</tr>
<tr>
<td><code>usbd_device_extra()</code></td>
<td>38</td>
</tr>
<tr>
<td><code>usbd_device_lookup()</code></td>
<td>39</td>
</tr>
<tr>
<td><code>usbd_disconnect()</code></td>
<td>40</td>
</tr>
<tr>
<td><code>usbd_endpoint_descriptor()</code></td>
<td>41</td>
</tr>
<tr>
<td><code>usbd_feature()</code></td>
<td>43</td>
</tr>
<tr>
<td><code>usbd_free()</code></td>
<td>45</td>
</tr>
<tr>
<td><code>usbd_free_urb()</code></td>
<td>46</td>
</tr>
<tr>
<td><code>usbd_get_frame()</code></td>
<td>47</td>
</tr>
<tr>
<td><code>usbd_hcd_ext_info()</code>, <code>usbd_hcd_info()</code></td>
<td>48</td>
</tr>
<tr>
<td><code>usbd_hub_descriptor()</code></td>
<td>50</td>
</tr>
<tr>
<td><code>usbd_interface_descriptor()</code></td>
<td>52</td>
</tr>
<tr>
<td><code>usbd_io()</code></td>
<td>54</td>
</tr>
<tr>
<td><code>usbd_languages_descriptor()</code></td>
<td>56</td>
</tr>
<tr>
<td><code>usbd_mphys()</code></td>
<td>58</td>
</tr>
<tr>
<td><code>usbd_open_pipe()</code></td>
<td>59</td>
</tr>
<tr>
<td><code>usbd_parse_descriptors()</code></td>
<td>61</td>
</tr>
<tr>
<td><code>usbd_pipe_device()</code></td>
<td>63</td>
</tr>
<tr>
<td><code>usbd_pipe_endpoint()</code></td>
<td>64</td>
</tr>
<tr>
<td><code>usbd_reset_device()</code></td>
<td>65</td>
</tr>
<tr>
<td><code>usbd_reset_pipe()</code></td>
<td>66</td>
</tr>
<tr>
<td><code>usbd_select_config()</code></td>
<td>67</td>
</tr>
<tr>
<td><code>usbd_select_interface()</code></td>
<td>68</td>
</tr>
<tr>
<td><code>usbd_setup_bulk()</code></td>
<td>70</td>
</tr>
<tr>
<td><code>usbd_setup_control()</code></td>
<td>72</td>
</tr>
<tr>
<td><code>usbd_setup_interrupt()</code></td>
<td>74</td>
</tr>
<tr>
<td><code>usbd_setup_isochronous()</code></td>
<td>76</td>
</tr>
<tr>
<td><code>usbd_setup_vendor()</code></td>
<td>78</td>
</tr>
<tr>
<td><code>usbd_status()</code></td>
<td>80</td>
</tr>
</tbody>
</table>
usbd_string() 82
usbd_topology(), usbd_topology_ext() 84
usbd_urb_status() 86
This chapter includes descriptions of the USB functions in alphabetical order, along with a listing of the functions arranged by category.

These functions are defined in the `libusbdi` library. Use the `-l usbdi` option to link against this library.

## Functions arranged by category

The USB functions may be grouped into these categories:

- Connection functions
- Memory-management functions
- I/O functions
- Pipe-management functions
- Configuration/interface functions
- Miscellaneous functions

### Connection functions

- `usbd_connect()`: Connect a client driver to the USB stack.
- `usbd_disconnect()`: Disconnect a client driver from the USB stack.
- `usbd_attach()`: Attach to a USB device.
- `usbd_detach()`: Detach from a USB device.

### Memory-management functions

- `usbd_alloc()`: Allocate memory area to use for data transfers.
- `usbd_free()`: Free memory allocated by `usbd_alloc()`.
- `usbd_mphys()`: Get the physical address of memory allocated by `usbd_alloc()`.
- `usbd_alloc_urb()`: Allocate a USB Request Block for subsequent URB-based operations.
- `usbd_free_urb()`: Free the URB allocated by `usbd_alloc_urb()`.

### I/O functions

- `usbd_setup_bulk()`: Set up a URB for a bulk data transfer.
- `usbd_setup_interrupt()`: Set up a URB for an interrupt transfer.
Functions arranged by category

```
usbd_setup_isochronous()

Set up a URB for an isochronous transfer.

usbd_setup_vendor()

Set up a URB for a vendor-specific transfer.

usbd_setup_control()

Set up a URB for a control transfer.

usbd_io()

Submit a previously set up URB to the USB stack.

usbd_feature()

Control a feature for a USB device.

usbd_descriptor()

Get or set USB descriptors.

usbd_status()

Get specific device status.
```

Pipe-management functions

```
usbd_open_pipe()

Initialize the pipe described by the device or endpoint descriptor.

usbd_close_pipe()

Close a pipe previously opened by the usbd_open_pipe() function.

usbd_reset_pipe()

Clear a stall condition on an endpoint identified by the pipe handle.

usbd_abort_pipe()

Abort all requests on a pipe.

usbd_pipe_device()

Retrieve the device associated with the pipe.

usbd_pipe_endpoint()

Retrieve the endpoint number associated with the pipe.
```

Configuration and interface functions

```
usbd_select_config()

Select the configuration for a USB device.

usbd_select_interface()

Select the interface for a USB device.
```

Miscellaneous and convenience functions

```
usbd_args_lookup()

Look up a driver’s command-line arguments.

usbd_configuration_descriptor()

Get the configuration descriptor for a specific configuration setting.
```
**usbd_device_lookup()**

Map the device instance identifier to an opaque device handle (from `usbd_attach()`).

**usbd_device_extra()**

Retrieve a pointer to the device-specific extra memory allocated by `usbd_attach()`.

**usbd_device_descriptor()**

Get the device descriptor for a specific device.

**usbd_endpoint_descriptor()**

Get the endpoint descriptor for a specific endpoint setting.

**usbd_get_frame()**

Get the current frame number and frame length for a device.

**usbd_hcd_ext_info(), usbd_hcd_info()**

Get information on the USB host controller and DDK library.

**usbd_hub_descriptor()**

Get the hub descriptor for a specific (hub) device.

**usbd_interface_descriptor()**

Get the interface descriptor for a specific interface setting.

**usbd_languages_descriptor()**

Get the table of supported LANGIDs for the given device.

**usbd_parse_descriptors()**

Parse device descriptors looking for a specific entry.

**usbd_reset_device()**

Reset a USB device.

**usbd_string()**

Get a string descriptor.

**usbd_urb_status()**

Return status information on a URB.

**usbd_topology(), usbd_topology_ext()**

Get the USB bus physical topology.
Abort all requests on a pipe

Synopsis:

```c
#include <sys/usbdi.h>

int usbd_abort_pipe( struct usbd_pipe *pipe );
```

Arguments:

- `pipe` An opaque handle returned by `usbd_open_pipe()`.

Library:

`libusbdi`

Description:

The `usbd_abort_pipe()` function aborts all requests on the specified pipe. You can use this function during an error condition (e.g. to abort a pending operation) or during normal operation (e.g. to halt an isochronous transfer).

Returns:

- EOK Success.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`usbd_open_pipe()`, `usbd_close_pipe()`, `usbd_pipe_endpoint()`, `usbd_reset_pipe()`
Synopsis:

```c
#include <sys/usbdi.h>

void *usbd_alloc( size_t size );
```

Arguments:

- `size` The size, in bytes, of the area to allocate.

Library:

`libusbdi`

Description:

The `usbd_alloc()` function allocates a memory area that can then be used for data transfers. You should use the memory area allocated by this function, because it’s allocated efficiently and because its physical address is quickly obtained via `usbd_mphys()`.

The `usbd_setup_*()` functions require `usbd_alloc()`’d data buffers.

To free the memory, use `usbd_free()`.

Returns:

A pointer to the start of the allocated memory, or NULL if there’s not enough memory.

Errors:

- ENOMEM Insufficient memory available.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

usbd_alloc_urb(), usbd_free(), usbd_free_urb(), usbd_mphys()
usbd_alloc_urb()
Allocate a USB Request Block for subsequent URB-based operations

Synopsis:

```c
#include <sys/usbdi.h>

struct usbd_urb *usbd_alloc_urb( struct usbd_urb *link );
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>link</td>
<td>Specifies multiple URBs linked together. <em>(Not yet implemented.)</em></td>
</tr>
</tbody>
</table>

Library:

```
libusbdi
```

Description:

The `usbd_alloc_urb()` function allocates a USB Request Block (URB) to be used for subsequent URB-based I/O transfers.

To free the block, use `usbd_free_urb()`.

Returns:

A pointer to the start of the allocated block, or NULL if there isn’t enough memory.

Errors:

```
ENOMEM Insufficient memory available.
```

Classification:

```
QNX Neutrino, QNX 4
```

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

```
usbd_alloc(), usbd_free(), usbd_free_urb(), usbd_mphys()
```
**Synopsis:**

```
#include <sys/usbdio.h>

void usbd_args_lookup(struct usbd_connection *connection,
                      int *argc,
                      char ***argv);
```

**Arguments:**

- `connection`: Identifies the USB stack (from `usbd_connect()`).

**Library:**

libusbdio

**Description:**

The `usbd_args_lookup()` function lets you look up a device driver's command-line arguments at insertion/attach time.

The command-line arguments are held in `argc` and `argv` within the `usbd_connect_parm` data structure. See `usbd_connect()` for details.

**Classification:**

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

- `usbd_configuration_descriptor()`, `usbd_connect()`, `usbd_device_lookup()`, `usbd_device_extra()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_interface_descriptor()`, `usbd_languages_descriptor()`, `usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
**Synopsis:**

```c
#include <sys/usbdi.h>

int usbd_attach( struct usbd_connection *connection,
                 usbd_device_instance_t *instance,
                 size_t extra,
                 struct usbd_device **device );
```

**Arguments:**

- `connection` An opaque handle that identifies the USB stack (from `usbd_connect()`).
- `instance` Describes which device you wish to attach to.
- `extra` The size of additional memory you’d like allocated with the device. You can use `usbd_device_extra()` later to get a pointer to this additional memory. Typically, the class driver would store various status/config/device-specific details in here (if needed).
- `device` An opaque handle used to identify the device in later calls.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_attach()` function to attach to a USB device. Typically, you do this out of the insertion callback (made when the device matched your filter), which will give you the `connection` and `instance` parameters involved. The insertion callback is prototyped as follows:

```c
void (*insertion)(struct usbd_connection *, usbd_device_instance_t *instance)
```

The `usbd_device_instance_t` structure looks like this:

```c
typedef struct usbd_device_instance {
    _uint8 path;
    _uint8 devno;
    _uint16 generation;
    usbd_device_ident_t ident;
    _uint32 config;
    _uint32 iface;
    _uint32 alternate;
} usbd_device_instance_t;
```
Looping

Another way to attach is to loop and attach to all devices (in which case you build the instance yourself). For example:

```c
for (busno = 0; busno < 10; ++busno) {
    for (devno = 0; devno < 64; ++devno) {
        memset(&instance, USBD_CONNECT_WILDCARD, sizeof(usbd_device_instance_t));
        instance.path = busno, instance.devno = devno;
        if (usbd_attach(connection, &instance, 0, &device) == EOK) {
            .......
        }
    }
}
```

The degree of “attachedness” depends on how you connected:

- If you specified insertion/removal callback functions, then you’ll get exclusive access to the device and can make I/O to it.
- If you didn’t use callbacks and you attached as in the loop above, you get shared access, so you can only read device configuration.

Returns:

- **EOK** Success.
- **ENODEV** Specified device doesn’t exist. If in a loop, then there’s nothing at that devno. If from a callback, then the device has since been removed.
- **EBUSY** A shared/exclusive conflict.
- **ENOMEM** No memory for internal device structures.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

`usbd_connect()`, `usbd_detach()`, `usbd_device_extra()`, `usbd_disconnect()`
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_close_pipe( struct usb管 *pipe );
```

Arguments:

`pipe` An opaque handle returned by `usbd_open_pipe()`.

Library:

`libusbdi`

Description:

You use the `usbd_close_pipe()` function to close a pipe that was previously opened via `usbd_open_pipe()`.

Returns:

- **EOK** Success.
- **EBUSY** Active or pending I/O.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`usbd_abort_pipe()`, `usbd_open_pipe()`, `usbd_pipe_endpoint()`, `usbd_reset_pipe()`
Synopsis:

```
#include <sys/usbdi.h>

usbd_configuration_descriptor_t
    *usbd_configuration_descriptor(
            struct usbd_device *device,
            _uint8 cfg,
            struct usbd_desc_node **node );
```

Arguments:

- **device**
  An opaque handle used to identify the USB device.
- **cfg**
  The device’s configuration identifier (`bConfigurationValue`).
- **node**
  Indicates the descriptor’s location for rooting future requests (e.g. interfaces of this configuration).

Library:

`libusbdi`

Description:

The `usbd_configuration_descriptor()` function lets you obtain the configuration descriptor for a specific configuration setting.

The `usbd_configuration_descriptor_t` structure looks like this:

```
typedef struct usbd_configuration_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint16 wTotalLength;
    _uint8 bNumInterfaces;
    _uint8 bConfigurationValue;
    _uint8 iConfiguration;
    _uint8 bmAttributes;
    _uint8 MaxPower;
} usbd_configuration_descriptor_t;
```

Returns:

A pointer to `usbd_configuration_descriptor_t` on success, or NULL on error.

Classification:

QNX Neutrino, QNX 4
### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_args_lookup()`, `usbd_device_lookup()`, `usbd_device_extra()`,  
`usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`,  
`usbd_hub_descriptor()`, `usbd_interface_descriptor()`, `usbd_languages_descriptor()`,  
`usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
usbd_connect()  
Connect a client driver to the USB stack

Synopsis:

```c
#include <sys/usbdi.h>

int usbd_connect( usbd_connect_parm_t *parm,
                 struct usbd_connection **connection );
```

Arguments:

- `parm`  Connection parameters describing how to connect to the USB stack and how you intend to operate with it.
- `connection`  An opaque handle returned on a successful connection; it’s used to pass into other routines to identify the connection.

Library:

`libusbdi`

Description:

You use the `usbd_connect()` function to connect to a USB device and to provide insertion/removal callbacks (in the `usbd_connect_parm_t` data structure).

Data structures

```c
typedef struct usbd_connect_parm {
    const char *path;
    _uint16 vusb;
    _uint16 vusbd;
    _uint32 flags;
    int argc;
    char **argv;
    _uint32 evtbufsz;
    usbd_device_ident_t *ident;
    usbd_funcs_t **funcs;
    _uint16 connect_wait
} usbd_connect_parm_t;
```

- `path`  Name of the stack (NULL means `/dev/io-usb/io-usb`, the default name).
- `vusb` and `vusbd`  Versions of the USB stack (USB _VERSION) and DDK (USBD _VERSION).
- `flags`  Currently none defined. Pass 0.
- `argc` and `argv`  Command-line arguments to the device driver that can be made available via `usbd_args_lookup()` at insertion/attach time.
- `evtbufsz`  Size of the event buffer used by the handler thread to buffer events from the USB stack. For the default size, pass 0.
ident A pointer to a `usbd_device_ident_t` structure that identifies the devices you’re interested in receiving insertion/removal callbacks for (a filter):

```c
typedef struct usbd_device_ident {
    _uint32 vendor;
    _uint32 device;
    _uint32 dclass;
    _uint32 subclass;
    _uint32 protocol;
} usbd_device_ident_t;
```

You can set the fields to `USBD_CONNECT_WILDCARD` or to an explicit value. You would typically make the `usbd_device_ident_t` structure be a filter for devices you support from this specific class driver.

funcs A pointer to a `usbd_funcs_t` structure that specifies the insertion/removal callbacks:

```c
typedef struct usbd_funcs {
    _uint32 nentries;
    void (*insertion)(struct usbd_connection *, usbd_device_instance_t *instance);
    void (*removal)(struct usbd_connection *, usbd_device_instance_t *instance);
    void (*event)(struct usbd_connection *, usbd_device_instance_t *instance,
                  _uint16 type);
} usbd_funcs_t;
```

The `usbd_funcs_t` structure includes the following members:

- **nentries** The number of entries in the structure. Set this to `_USBDI_NFUNCTIONS`.
- **insertion** The function to call when a device that matches the defined filter is detected.
- **removal** The function to call when a device is removed.
- **event** A future extension for various other event notifications (e.g. bandwidth problems).

By passing NULL as the `usbd_funcs`, you’re saying that you’re not interested in receiving dynamic insertion/removal notifications, which means that you won’t be a fully operational class driver. No asynchronous I/O will be allowed, no event thread, etc. This approach is taken, for example, by the `usb` display utility.

**connect_wait** A value (in seconds) or `USBD_CONNECT_WAIT`.

**Returns:**

- **EOK** Success.
- **EPROGMISMATCH** Versionitis.
- **ENOMEM** No memory for internal connect structures.
ESRCH    USB server not running.
EACCESS  Permission denied to USB server.
EAGAIN   Can’t create async/callback thread.

Examples:

A class driver (in its main(), probably) for a 3COM Ethernet card might connect like this:

```c
usbd_device_ident_t interest = {
    USB_VENDOR_3COM,
    USB_PRODUCT_3COM_3C19250,
    USBD_CONNECT_WILDCARD,
    USBD_CONNECT_WILDCARD,
    USBD_CONNECT_WILDCARD,
};
usbd_funcs_t funcs = {
    _USBDI_NFUNCS,
    insertion,
    removal,
    NULL
};
usbd_connect_parm_t cparsms = {
    NULL,
    USB_VERSION,
    USBD_VERSION,
    0,
    argc,
    argv,
    0,
    &interest,
    &funcs
};
struct usbd_connection *connection;
int error;

error = usbd_connect(&cparsms, &connection);
```

Classification:

QNX Neutrino, QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Caveats:

The `usbd_connect()` function creates a thread on your behalf that’s used by the library to monitor the USB stack for device insertion or removal. Since your insertion and removal callback functions are called by this new thread, you must ensure that any common resources used between that thread and any other thread(s) in your class driver are properly protected (e.g. via a mutex).

See also:

`usbd_args_lookup()`, `usbd_attach()`, `usbd_detach()`, `usbd_disconnect()`
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_descriptor( struct usbd_device *device,
                      int set,
                      _uint8 type,
                      _uint16 rtype,
                      _uint8 index,
                      _uint16 langid,
                      _uint8 *desc,
                      size_t len );
```

Arguments:

- **device**: An opaque handle used to identify the USB device.
- **set**: A flag that says to either get or set a descriptor.
- **type**: Type of descriptor (e.g. USB_DESC_DEVICE, USB_DESC_CONFIGURATION, USB_DESC_STRING, USB_DESC_HUB).
- **rtype**: Type of request (e.g. USB_RECIPIENT_DEVICE, USB_RECIPIENT_INTERFACE, USB_RECIPIENT_ENDPOINT, USB_RECIPIENT_OTHER, USB_TYPE_STANDARD, USB_TYPE_CLASS, USB_TYPE_VENDOR).
- **index**: This varies, depending on the request. It’s used for passing a parameter to the device.
- **langid**: Identifies the language supported in strings (according to the LANGID table).
- **desc**: Pointer at buffer to put descriptors.
- **len**: The length of the data transfer in bytes.

Library:

- **libusbdi**

Description:

The `usbd_descriptor()` function lets you obtain the USB descriptors.

Returns:

- **EMSGSIZE**: Buffer too small for descriptor.
- **ENOMEM**: No memory for URB.
ENODEV       Device was removed.
EIO          I/O error on USB device.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

usbd_feature(), usbd_io(), usbd_parse_descriptors(), usbd_setup_bulk(), usbd_setup_control(), usbd_setup_interrupt(), usbd_setup_isochronous(), usbd_setup_vendor(), usbd_status()
usbd_detach()

Detach from the USB device

Synopsis:

```c
#include <sys/usbd.h>

int usbd_detach( struct usbd_device *device );
```

Arguments:

device An opaque handle from usbd_attach().

Library:

libusbd

Description:

You use the usbd_detach() function to disconnect from a USB device that you previously had attached to via usbd_attach().

The usbd_detach() function automatically closes any pipes previously opened via usbd_open_pipe().

Returns:

- EOK Success.
- EBUSY I/O pending on the device.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats:

Don’t try to detach if there’s I/O pending on the device. If there is, usbd_detach() will fail.
See also:

usbd_attach(), usbd_close_pipe(), usbd_connect(), usbd_disconnect(),
usbd_open_pipe()
**Synopsis:**

```c
#include <sys/usbdi.h>

usbd_device_descriptor_t *
usbd_device_descriptor(
    struct usbd_device *device,
    struct usbd_desc_node **node);
```

**Arguments:**

- **device**: A handle obtained by calling `usbd_attach()`.
- **node**: The address of a pointer to a `usbd_device_descriptor_t` structure where the function stores the device descriptor.

**Library:**

- `libusbdi`

**Description:**

The `usbd_device_descriptor()` function lets you obtain the device descriptor for a specific device.

The `node` parameter tells you where a descriptor was found to root future requests from (e.g. configurations of the device).

The `usbd_device_descriptor_t` structure looks like this:

```c
typedef struct usbd_device_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint16 bcdUSB;
    _uint8 bDeviceClass;
    _uint8 bDeviceSubClass;
    _uint8 bDeviceProtocol;
    _uint8 bMaxPacketSize0;
    _uint16 idVendor;
    _uint16 idProduct;
    _uint16 bcdDevice;
    _uint8 iManufacturer;
    _uint8 iProduct;
    _uint8 iSerialNumber;
    _uint8 bNumConfigurations;
} usbd_device_descriptor_t;
```

**Returns:**

A pointer to `usbd_device_descriptor_t` on success, or NULL on error.
Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

usbd_args_lookup(), usbd_configuration_descriptor(), usbd_device_lookup(), usbd_device_extra(), usbd_endpoint_descriptor(), usbd_hcd_info(), usbd_hub_descriptor(), usbd_interface_descriptor(), usbd_languages_descriptor(), usbd_parse_descriptors(), usbd_string(), usbd_urb_status()
**Synopsis:**

```c
#include <sys/usbdi.h>

void *usbd_device_extra( struct usbd_device *device );
```

**Arguments:**

- `device`: A handle obtained by calling `usbd_attach()`.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_device_extra()` function to get a pointer to the additional memory allocated via the `extra` parameter in `usbd_attach()`.

**Returns:**

A pointer to the additional memory, or NULL if no device-specific memory was allocated by `usbd_attach()`.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<table>
<thead>
<tr>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_args_lookup()`, `usbd_attach()`, `usbd_configuration_descriptor()`, `usbd_device_lookup()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_interface_descriptor()`, `usbd_languages_descriptor()`, `usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
**Synopsis:**

```c
#include <sys/usbdi.h>

struct usbd_device *usbd_device_lookup(
    struct usbd_connection *connection,
    usbd_device_instance_t *instance );
```

**Arguments:**

- `connection` A handle obtained by calling `usbd_connect()`.
- `instance` The device instance identifier obtained by calling `usbd_attach()`.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_device_lookup()` function to map the device instance identifier to an opaque device handle. This is typically required in the removal callback.

**Returns:**

An opaque device handle, or NULL.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

- Cancellation point: No
- Interrupt handler: No
- Signal handler: No
- Thread: Yes

**See also:**

`usbd_args_lookup()`, `usbd_attach()`, `usbd_configuration_descriptor()`, `usbd_device_extra()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_interface_descriptor()`, `usbd_languages_descriptor()`, `usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
**Synopsis:**

```c
#include <sys/usbdi.h>

int usbd_disconnect( struct usbd_connection *connection );
```

**Arguments:**

- `connection` A handle for the USB stack, obtained by calling `usbd_connect()`.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_disconnect()` to disconnect a client driver that had been previously connected to the USB stack via the `usbd_connect()` function.

The `usbd_disconnect()` function automatically closes any pipes previously opened via `usbd_attach()`.

**Returns:**

- `EOK` Success.

**Classification:**

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_attach()`, `usbd_connect()`, `usbdDetach()`
Synopsis:

```c
#include <sys/usbdi.h>

usbd_endpoint_descriptor_t
*usbd_endpoint_descriptor(
    struct usbd_device *device,
    _uint8 config,
    _uint8 iface,
    _uint8 alt,
    _uint8 endpoint,
    struct usbd_desc_node **node );
```

Arguments:

- `device` An opaque handle used to identify the USB device.
- `config` Configuration identifier (`bConfigurationValue`).
- `iface` Interface identifier (`bInterfaceNumber`).
- `alt` Alternate identifier (`bAlternateSetting`).
- `endpoint` Endpoint identifier (`bEndpointAddress`).
- `node` Indicates the descriptor’s location for rooting future requests.

Library:

`libusbdi`

Description:

The `usbd_endpoint_descriptor()` function lets you obtain the endpoint descriptor for a specific endpoint on a configuration/interface.

The `endpoint_descriptor_t` structure looks like this:

```c
typedef struct usbd_endpoint_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint8 bEndpointAddress;
    _uint8 bmAttributes;
    _uint8 bInterval;
    _uint16 wMaxPacketSize;
} usbd_endpoint_descriptor_t;
```

Returns:

A pointer to `usbd_endpoint_descriptor_t` on success, or NULL on error.
usbd_endpoint_descriptor()

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

usbd_args_lookup(), usbd_configuration_descriptor(), usbd_device_lookup(),
usbd_device_extra(), usbd_device_descriptor(), usbd_hcd_info(),
usbd_hub_descriptor(), usbd_interface_descriptor(), usbd_languages_descriptor(),
usbd_parse_descriptors(), usbd_string(), usbd_urb_status()
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_feature( struct usbd_device *device,
                  int set,
                  _uint16 feature,
                  _uint16 rtype,
                  _uint16 index );
```

Arguments:

- `device` An opaque handle used to identify the USB device.
- `set` Set or clear a feature on the USB device.
- `feature` A specific feature on the device.
- `rtype` Type of request (e.g. USB_RECIPIENT_DEVICE, USB_RECIPIENT_INTERFACE, USB_RECIPIENT_ENDPOINT, USB_RECIPIENT_OTHER, USB_TYPE_STANDARD, USB_TYPE_CLASS, USB_TYPE_VENDOR).
- `index` This varies, depending on the request. It’s used for passing a parameter to the device.

Library:

- `libusbdi`

Description:

The `usbd_feature()` function lets you control a specific feature on a USB device.

Returns:

- `EOK` Success.
- `ENOMEM` No memory for URB.
- `ENODEV` Device was removed.
- `EIO` I/O error on USB device.

Classification:

- QNX Neutrino, QNX 4
usbd_feature()

Safety

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt</td>
<td>No</td>
</tr>
<tr>
<td>Signal</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

usbd_descriptor(), usbd_iot(), usbd_setup_bulk(), usbd_setup_control(), usbd_setup_interrupt(), usbd_setup_isochronous(), usbd_setup_vendor(), usbd_status()
**Synopsis:**

```c
#include <sys/usbdi.h>

void usbd_free( void* ptr );
```

**Arguments:**

- `ptr` A pointer to the memory area to be freed.

**Library:**

`libusbd`i

**Description:**

The `usbd_free()` function frees the memory allocated by `usbd_alloc()`. The function deallocates the memory area specified by `ptr`, which was previously returned by a call to `usbd_mphys()`.

It’s safe to call `usbd_free()` with a NULL `ptr`.

**Returns:**

- `EOK` Success.

**Classification:**

QNX Neutrino, QNX 4

### Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_alloc()`, `usbd_alloc_urb()`, `usbd_free_urb()`, `usbd_mphys()`
**usbd_free_urb()**

*Free the USB Request Block allocated by usbd_alloc_urb()*

**Synopsis:**

```c
#include <sys/usbdi.h>

struct usbd_urb *usbd_free_urb( struct usbd_urb *urb );
```

**Arguments:**

*urb*  A pointer to the URB to be freed.

**Library:**

libusbdi

**Description:**

The `usbd_free_urb()` function frees the memory allocated by `usbd_alloc_urb()`.

**Returns:**

EOK  Success.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_alloc()`, `usbd_alloc_urb()`, `usbd_free()`, `usbd_mphys()`
**usbd_get_frame()**

*Get the current frame number and frame length for a device*

**Synopsis:**

```c
int usbd_get_frame( struct usdb_device *device,
           _int32 *fnum,
           _int32 *flen );
```

**Arguments:**

- `device`  The handle for the device, obtained by calling `usbd_attach()`.
- `fnum`    If non-NULL, this is set to the frame number.
- `flen`    If non-NULL, this is set to the frame length.

**Library:**

`libusbdi`

**Description:**

This function gets the current frame number and frame length for the specified device.

**Returns:**

- `EOK`     Success.
- `ENODEV`  The device has been removed.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<table>
<thead>
<tr>
<th>安全特性</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>取消请求</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>中断处理程序</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>信号处理程序</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>线程</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See also:**

`usbd_attach()`
Synopsis:

#include <sys/usbd.h>

int usbd_hcd_ext_info( struct usbd_connection *connection,
                       _uint32 cindex,
                       usbd_hcd_info_t *info );

int usbd_hcd_info( struct usbd_connection *connection,
                    usbd_hcd_info_t *info );

Arguments:

connection The handle for the connection to the USB stack, obtained by calling usbd_connect().

cindex (usbd_hcd_ext_info() only) The index of the host controller.

info A pointer to a usbd_hcd_info_t data structure that this function fills in.

Library:

libusbd

Description:

You can use the usbd_hcd_ext_info() or usbd_hcd_info() function to obtain information from the USB host controller and DDK library.

If your system has more than one USB chip, you can call usbd_hcd_ext_info() to get information about a specific one. The usbd_hcd_info() function gets information about the first USB chip; calling it is the same as calling usbd_hcd_ext_info() with a cindex argument of 0.

The usbd_hcd_info_t structure is defined as follows:

typedef struct usbd_hcd_info {
    _uint16 vusb1
    _uint16 vusbd1
    char controller[8];
    _uint32 capabilities;
    _uint8 ndev;
    _uint8 cindex;
    _uint16 vhcd;
    _uint32 max_td_io;
    _uint8 reserved[12];
} usbd_hcd_info_t;

It contains at least the following:

vusb The version number of the USB stack.
vusbd
controller

capabilities
ndev

cindex

vhcd

max_tid_io

The version number of the USB DDK.
The name of the USB host controller.
The capabilities of the USB host controller.
The number of devices currently connected.
The index of the host controller.
The version number of the USB HCD.
The maximum number of bytes per HC TD.

Returns:

EOK    Success.

Classification:

QNX Neutrino, QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

usbd_args_lookup(), usbd_configuration_descriptor(), usbd_device_lookup(), usbd_device_extra(), usbd_device_descriptor(), usbd_endpoint_descriptor(), usbd_hub_descriptor(), usbd_interface_descriptor(), usbd_languages_descriptor(), usbd_parse_descriptors(), usbd_string(), usbd_urb_status()
**Synopsis:**

```c
#include <sys/usbd.h>

usbd_hub_descriptor_t *usbd_hub_descriptor(
    struct usbd_device *device,
    struct usbd_desc_node **node);
```

**Arguments:**

- **device** An opaque handle used to identify the USB device.
- **node** Indicates the descriptor’s location for rooting future requests.

**Library:**

`libusbd`  

**Description:**

The `usbd_hub_descriptor()` function lets you obtain a hub descriptor.

The `usbd_hub_descriptor_t` data structure looks like this:

```c
typedef struct usbd_hub_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint8 bNbrPorts;
    _uint16 wHubCharacteristics;
    _uint8 bPwrOn2PwrGood;
    _uint8 bHubContrCurrent;
    _uint8 DeviceRemovable[1];
    _uint8 PortPwrCtrlMask[1];
} usbd_hub_descriptor_t;
```

**Returns:**

A pointer to `usbd_hub_descriptor_t` on success, or NULL on error.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

usbd_args_lookup(), usbd_configuration_descriptor(), usbd_device_lookup(), usbd_device_extra(), usbd_device_descriptor(), usbd_endpoint_descriptor(), usbd_hcd_info(), usbd_interface_descriptor(), usbd_languages_descriptor(), usbd_parse_descriptors(), usbd_string(), usbd_urb_status()
Synopsis:

#include <sys/usbdi.h>

usbd_interface_descriptor_t
*usbd_interface_descriptor(
    struct usbd_device *device,
    _uint8 cfg,
    _uint8 ifc,
    _uint8 alt,
    struct usbdDescNode **node);

Arguments:

device An opaque handle used to identify the USB device.
cfg The device’s configuration identifier (bConfigurationValue).
ifc Interface identifier (bInterfaceNumber).
alt Alternate identifier (bAlternateSetting).
node Indicates the descriptor’s location for rooting future requests (e.g.
endpoints of this interface).

Library:

libusbdi

Description:

The usbd_interface_descriptor() function lets you obtain the interface descriptor for a
specific interface setting.

The usbd_interface_descriptor_t structure looks like this:

typedef struct usbd_interface_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint8 bInterfaceNumber;
    _uint8 bAlternateSetting;
    _uint8 bNumEndpoints;
    _uint8 bInterfaceClass;
    _uint8 bInterfaceSubClass;
    _uint8 bInterfaceProtocol;
    _uint8 iInterface;
} usbd_interface_descriptor_t;
Returns:

A pointer to `usbd_interface_descriptor_t` on success, or NULL on error.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`usbd_args_lookup()`, `usbd_configuration_descriptor()`, `usbd_device_lookup()`, `usbd_device_extra()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_languages_descriptor()`, `usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
Submit a previously set up URB to the USB stack

Synopsis:

```c
#include <sys/usbdi.h>

int usbd_io( struct usbd_urb *urb,
             struct usbd_pipe *pipe,
             void (*func)(struct usbd_urb *,
                          struct usbd_pipe *, void *),
             void *handle,
             _uint32 timeout );
```

Arguments:

- **urb**: A pointer to a USB Request Block.
- **pipe**: An opaque handle returned by `usbd_open_pipe()`.
- **func**: Callback at I/O completion, given URB, pipe, plus handle.
- **handle**: User data.
- **timeout**: A value (in milliseconds) or USBD_TIME_DEFAULT or USBD_TIME_INFINITY.

Library:

`libusbdi`

Description:

This routine submits a previously set up URB to the USB stack. The URB would have been set up from one of these functions:

- `usbd_setup_bulk`
- `usbd_setup_control`
- `usbd_setup_interrupt`
- `usbd_setup_isochronous`
- `usbd_setup_vendor`

For this release of the USB DDK, vendor requests are synchronous only. Therefore, the `func` parameter in `usbd_io()` must be NULL.

The `usbd_io()` function is the one that actually makes the data transfer happen; the setup functions simply set up the URB for the data transfer.
Returns:

- **EBADF** Improper `usbd_connect()` call.
- **EINVAL** Improper `usbd_connect()` call.
- **ENODEV** Device was removed.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

See also:

- `usbd_descriptor()`, `usbd_feature()`, `usbd_setup_control()`, `usbd_setup_bulk()`, `usbd_setup_interrupt()`, `usbd_setup_isochronous()`, `usbd_setup_vendor()`, `usbd_status()`
**Synopsis:**

```c
#include <sys/usbdi.h>

usbd_string_descriptor_t *usbd_languages_descriptor(
    struct usbd_device *device,
    struct usbd_desc_node **node);
```

**Arguments:**

- `device` An opaque handle used to identify the USB device.
- `node` Indicates the descriptor’s location for rooting future requests.

**Library:**

- `libusbdi`

**Description:**

The `usbd_languages_descriptor()` function lets you obtain the table of supported language IDs for the device.

The `usbd_string_descriptor_t` structure looks like this:

```c
typedef struct usbd_string_descriptor {
    _uint8 bLength;
    _uint8 bDescriptorType;
    _uint16 bString[1];
} usbd_string_descriptor_t;
```

**Returns:**

A pointer `usbd_string_descriptor_t` on success, or NULL on error.

**Classification:**

- QNX Neutrino, QNX 4

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

`usbd_args_lookup()`, `usbd_configuration_descriptor()`, `usbd_device_lookup()`,
`usbd_device_extra()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`,
`usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_interface_descriptor()`,
`usbd_parse_descriptors()`, `usbd_string()`, `usbd_urb_status()`
**usbd_mphys()**

Get the physical address of memory allocated by `usbd_alloc()`

### Synopsis:

```c
#include <sys/usbdi.h>

paddr_t usbd_mphys( const void *ptr );
```

### Arguments:

- `ptr` A pointer to the block of memory.

### Library:

`libusbdi`

### Description:

The `usbd_mphys()` function obtains the physical address used by `usbd_alloc()` to allocate memory for a data transfer.

### Returns:

Physical address.

### Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### See also:

`usbd_alloc()`, `usbd_alloc_urb()`, `usbd_free()`, `usbd_free_urb()`, `usbd_mphys()`
Synopsis:

```c
#include <sys/usbd.h>

int usbd_open_pipe( struct usbd_device *device,
                     usbd_descriptors_t *desc,
                     struct usbd_pipe **pipe );
```

Arguments:

- `device` - An opaque handle used to identify the USB device.
- `desc` - A pointer to the device or endpoint descriptor that was returned from `usbd_parse_descriptors()`.
- `pipe` - An opaque handle returned by `usbd_open_pipe()`.

Library:

`libusbd`  

Description:

You use the `usbd_open_pipe()` function to initialize the pipe described by the endpoint descriptor.

Returns:

- `EOK` - Success.
- `EINVAL` - The descriptor isn’t a device or endpoint.
- `ENOMEM` - No memory for internal pipe structures.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>
See also:

usbd_abort_pipe(), usbd_close_pipe(), usbd_pipe_endpoint(), usbd_reset_pipe()
Synopsis:

```c
#include <sys/usbdi.h>

usbd_descriptors_t *usbd_parse_descriptors(
    struct usbd_device *device,
    struct usbd_desc_node *root,
    _uint8 type,
    int index,
    struct usbd_desc_node **node);
```

Arguments:

- **device**: The opaque handle for the device whose descriptors you want to search.
- **root**: Where in the tree to begin parsing (pass NULL to start at the base).
- **type**: The type of descriptor to find (USB_DESC_*), or 0 to match any type.
- **index**: The occurrence of the descriptor that you want to find.
- **node**: A pointer to a location where the function stores a pointer to the descriptor that it found. You can use this as the root for future requests.

Library:

- libusbdi

Description:

When you call it the first time, the `usbd_parse_descriptors()` function loads all the descriptors from the USB device:

- device
- configuration
- interface
- endpoint
- hub
- string

The function uses `usbd_descriptor()` to get each raw USB descriptor. The data is then endian-ized, made alignment-safe, and built into an in-memory tree structure to facilitate future parsing requests.

Each node in this tree is a `struct usbd_desc_node`. The `root` parameter lets you say where in the tree to begin parsing (NULL is base). The `node` parameter tells you where a descriptor was found to root future requests from.

The tree looks like this:
Any vendor-specific or class-specific descriptors that are embedded into the standard descriptor output are also inserted into this tree at the appropriate point.

Although a descriptor for endpoint 0 (control) isn’t present on the wire, one is constructed and placed in the tree (to simplify enumeration within the class driver).

You use `type` for specifying the type of descriptor to find; `index` is the n-th occurrence. Note that type 0 will match any descriptor type; you can use it to retrieve any embedded class or vendor-specific descriptors if you don’t know their type.

Here’s an example that will walk all endpoints for an interface:

```c
for (eix = 0; (desc = usbd_parse_descriptors(device, ifc, USB_DESC_ENDPOINT, eix, &ept)) != NULL; ++eix) {
    //
}
```

where `ifc` is the appropriate (INTERFACE) node (found by a previous call to `usbd_parse_descriptors()` or `usbd_interface_descriptor()`).

Returns:

A pointer to the descriptor on success, or NULL on error.

Classification:

QNX Neutrino, QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`usbd_args_lookup()`, `usbd_configuration_descriptor()`, `usbd_descriptor()`, `usbd_device_lookup()`, `usbd_device_extra()`, `usbd_device_descriptor()`, `usbd_endpoint_descriptor()`, `usbd_hcd_info()`, `usbd_hub_descriptor()`, `usbd_interface_descriptor()`, `usbd_languages_descriptor()`, `usbd_string()`, `usbd_urb_status()`
**Synopsis:**

```c
#include <sys/usbdi.h>

struct usbd_device*
usbd_pipe_device( struct usbd_pipe *pipe );
```

**Arguments:**

- `pipe` An opaque handle returned by `usbd_open_pipe()`.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_pipe_device()` to retrieve the device associated with `pipe`.

**Returns:**

A pointer to a `usbd_device` structure that describes the device.

**Classification:**

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_abort_pipe(), usbd_open_pipe(), usbd_close_pipe(), usbd_reset_pipe()`
usbd_pipe_endpoint()

Retrieve the endpoint number associated with the pipe

Synopsis:

```c
#include <sys/usbdi.h>

_uint32 usbd_pipe_endpoint( struct usbd_pipe *pipe );
```

Arguments:

pipe An opaque handle returned by usbd_open_pipe().

Library:

libusbdi

Description:

You use the `usbd_pipe_endpoint()` to retrieve the endpoint number associated with `pipe`.

Returns:

A pipe/endpoint number.

Classification:

QNX Neutrino, QNX 4

Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

`usbd_abort_pipe()`, `usbd_open_pipe()`, `usbd_close_pipe()`, `usbd_reset_pipe()`
**usbd_reset_device()**

Reset a USB device

## Synopsis:

```c
#include <sys/usbdi.h>

int usbd_reset_device( struct usbd_device *device );
```

## Arguments:

- **device**
  The handle of a device.

## Library:

- **libusbdi**

## Description:

You use the `usbd_reset_device()` function to reset the specified `device`.

## Returns:

- **EOK**
  Success.
- **ENODEV**
  Device was removed.

## Classification:

- **QNX Neutrino, QNX 4**

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

## See also:

- `usbd_attach()`, `usbd_connect()`
**usbd_reset_pipe()**

Clear a stall condition on an endpoint identified by the pipe handle

**Synopsis:**

```c
#include <sys/usbdi.h>

int usbd_reset_pipe( struct usbd_pipe *pipe );
```

**Arguments:**

- `pipe`  
  An opaque handle returned by `usbd_open_pipe()`.

**Library:**

`libusbdi`

**Description:**

You use the `usbd_reset_pipe()` function to clear a stall condition on an endpoint identified by the `pipe` handle.

**Returns:**

- `EOK` Success.
- `ENOMEM` No memory for URB.
- `ENODEV` Device was removed.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_abort_pipe() usbd_open_pipe(), usbd_close_pipe(), usbd_pipe_endpoint()`.
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_select_config( struct usbd_device *device, _uint8 cfg );
```

Arguments:

- `device` An opaque handle used to identify the USB device.
- `cfg` The device’s configuration identifier (bConfigurationValue).

Library:

- libusbdi

Description:

You use the `usbd_select_config()` function to select the configuration for a USB device.

Returns:

- EOK Success.
- ENOMEM No memory for URB.
- ENODEV Device was removed.

Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

- `usbd_select_interface()`
Synopsis:

    #include <sys/usbd.h>

    int usbd_select_interface( struct usbd_device *device,
                    _uint8 ifc,
                    _uint8 alt );

Arguments:

    device  An opaque handle used to identify the USB device.
    ifc     Interface identifier (bInterfaceNumber).
    alt     Alternate identifier (bAlternateSetting).

Library:

    libusbd

Description:

You use the `usbd_select_interface()` function to select the interface for a USB device.

Returns:

    EOK        Success.
    ENOMEM     No memory for URB.
    ENODEV     Device was removed.

Classification:

    QNX Neutrino, QNX 4

    Safety
             Cancellation point | Yes
                        Interrupt handler | No
                        Signal handler    | No
                        Thread             | Yes
See also:

`usbd_select_config()`
usbd_setup_bulk()

Set up a URB for a bulk data transfer

Synopsis:

```c
#include <sys/usbdi.h>

int usbd_setup_bulk( struct usbd_urb *urb,
                     _uint32 flags,
                     void *addr,
                     _uint32 len );
```

Arguments:

- `urb` An opaque handle (from `usbd_alloc_urb()`).
- `flags` One of the following:
  - `URB_DIR_IN`—specify incoming (device-to-PC) transfer.
  - `URB_DIR_OUT`—specify outgoing (PC-to-device) transfer.
  - `URB_DIR_NONE`—don’t specify the direction.
  
  You can optionally OR in the following:
  - `URB_SHORT_XFER_OK`—allow short transfers.
- `addr` The address for the start of the transfer. You must use the buffer allocated by `usbd_alloc()`.
- `len` The length (in bytes) of the data transfer.

Library:

`libusbdi`

Description:

This routine sets up a URB for a bulk data transfer.

Returns:

- `EOK` Success.

Classification:

QNX Neutrino, QNX 4

Safety

<table>
<thead>
<tr>
<th>Cancellation point</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
</tbody>
</table>

continued…
<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Caveats:**

To ensure that the correct physical address will be used, you must use the buffer allocated by `usbd_alloc()` for the `addr` parameter.

**See also:**

`usbd_descriptor()`, `usbd_feature()`, `usbd_io()`, `usbd_setup_control()`, `usbd_setup_interrupt()`, `usbd_setup_isochronous()`, `usbd_setup_vendor()`, `usbd_status()`
**usbd_setup_control()**

Set up a URB for a control transfer

---

This function isn’t currently implemented. To set up a URB for a control transfer, use `usbd_setup_vendor()` instead.

### Synopsis:

```c
#include <sys/usbdi.h>

usbd_setup_control( struct usbd_urb *urb,
                    _uint32 flags,
                    _uint16 request,
                    _uint16 rtype,
                    _uint16 value,
                    _uint16 index,
                    void *addr,
                    _uint32 len );
```

### Arguments:

- **urb**  
  An opaque handle (from `usbd_alloc_urb()`).

- **flags**  
  One of the following:
  - URB_DIR_IN—specify incoming (device-to-PC) transfer.
  - URB_DIR_OUT—specify outgoing (PC-to-device) transfer.
  - URB_DIR_NONE—don’t specify the direction.
  
  You can optionally OR in the following:
  - URB_SHORT_XFER_OK—allow short transfers.

- **request**  
  A device-specific request.

- **rtype**  
  The type of request; one of the following:
  - USB_RECIPIENT_DEVICE
  - USB_RECIPIENT_INTERFACE
  - USB_RECIPIENT_ENDPOINT
  - USB_RECIPIENT_OTHER
  
  ORed with one of the following:
  - USB_TYPE_STANDARD
  - USB_TYPE_CLASS
  - USB_TYPE_VENDOR

- **value**  
  This varies, depending on the request. It’s used for passing a parameter to the device.

- **index**  
  This varies, depending on the request. It’s used for passing a parameter to the device.
**usbd_setup_control()**

*addr* The address for the start of the transfer. You *must* use the buffer allocated by *usbd_alloc()*.

*len* The length (in bytes) of the data transfer.

**Library:**

libusbdi

**Description:**

This routine sets up a URB for a control transfer.

**Returns:**

EOK Success.

**Classification:**

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
</tr>
<tr>
<td>Interrupt handler</td>
</tr>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**Caveats:**

To ensure that the correct physical address will be used, you *must* use the buffer allocated by *usbd_alloc()* for the *addr* parameter.

**See also:**

`usbd_descriptor()`, `usbd_feature()`, `usbd_iol()`, `usbd_setup_bulk()`, `usbd_setup_interrupt()`, `usbd_setup_isochronous()`, `usbd_setup_vendor()`, `usbd_status()`
Set up a URB for an interrupt transfer

Synopsis:

```
#include <sys/usbdi.h>

int usbd_setup_interrupt( struct usbd_urb *urb,
    _uint32 flags,
    void *addr,
    _uint32 len );
```

Arguments:

- `urb` An opaque handle (from `usbd_alloc_urb()`).
- `flags` One of the following:
  - URB_DIR_IN—specify incoming (device-to-PC) transfer.
  - URB_DIR_OUT—specify outgoing (PC-to-device) transfer.
  - URB_DIR_NONE—don’t specify the direction.
  
  You can optionally OR in the following:
  - URB_SHORT_XFER_OK—allow short transfers.
- `addr` The address for the start of the transfer. You must use the buffer allocated by `usbd_alloc()`.
- `len` The length (in bytes) of the data transfer.

Library:

`libusbdi`

Description:

This routine sets up a URB for an interrupt transfer.

Returns:

- EOK Success.

Classification:

QNX Neutrino, QNX 4

Safety

- Cancellation point No
- Interrupt handler No

continued…
**usbd_setup_interrupt()**

<table>
<thead>
<tr>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
</tr>
<tr>
<td>Thread</td>
</tr>
</tbody>
</table>

**See also:**

usbd_setup_bulk(), usbd_setup_control(), usbd_setup_isochronous(), usbd_setup_vendor()
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_setup_isochronous( struct usbd_urb *urb,
    _uint32 flags,
    _int32 frame,
    void *addr,
    _uint32 len );
```

Arguments:

- `urb`: An opaque handle (from `usbd_alloc_urb()`).
- `flags`: One of the following:
  - `URB_DIR_IN`—specify incoming (device-to-PC) transfer.
  - `URB_DIR_OUT`—specify outgoing (PC-to-device) transfer.
  - `URB_DIR_NONE`—don’t specify the direction.
  You can optionally OR in either or both of the following:
  - `URB_ISOCH_ASAP`—allow transfer as soon as possible (overrides `frame`).
  - `URB_SHORT_XFER_OK`—allow short transfers.
- `frame`: The device frame number. This is ignored if `URB_ISOCH_ASAP` is set.
- `addr`: The address for the start of the transfer. You must use the buffer allocated by `usbd_alloc()`.
- `len`: The length (in bytes) of the data transfer.

Library:

`libusbdi`

Description:

This routine sets up a URB for an isochronous transfer.

Returns:

- `EOK`: Success.

Classification:

QNX Neutrino, QNX 4
**usbd_setup_isochronous()**

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_descriptor()`, `usbd_feature()`, `usbd_io()`, `usbd_setup_bulk()`, `usbd_setup_control()`, `usbd_setup_interrupt()`, `usbd_setup_vendor()`, `usbd_status()`
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_setup_vendor( struct usbd_urb *urb,
                      _uint32 flags,
                      _uint16 request,
                      _uint16 rtype,
                      _uint16 value,
                      _uint16 index,
                      void *addr,
                      _uint32 len );
```

Arguments:

- **urb**: An opaque handle (from `usbd_alloc_urb()`).
- **flags**: One of the following:
  - `URB_DIR_IN`—specify incoming (device-to-PC) transfer.
  - `URB_DIR_OUT`—specify outgoing (PC-to-device) transfer.
  - `URB_DIR_NONE`—don’t specify the direction.
  You can optionally OR in the following:
  - `URB_SHORT_XFER_OK`—allow short transfers.
- **request**: A device-specific request.
- **rtype**: The type of request; one of the following:
  - `USB_RECIPIENT_DEVICE`
  - `USB_RECIPIENT_INTERFACE`
  - `USB_RECIPIENT_ENDPOINT`
  - `USB_RECIPIENT_OTHER`
  ORed with one of the following:
  - `USB_TYPE_STANDARD`
  - `USB_TYPE_CLASS`
  - `USB_TYPE_VENDOR`
- **value**: This varies, depending on the request. It’s used for passing a parameter to the device.
- **index**: This varies, depending on the request. It’s used for passing a parameter to the device.
- **addr**: The address for the start of the transfer. You *must* use the buffer allocated by `usbd_alloc()`.
- **len**: The length (in bytes) of the data transfer.
Library: libusbd

Description: This routine sets up a URB for a vendor-specific transfer.

For this release of the USB DDK, vendor requests are synchronous only. Therefore, the `func` parameter in `usbd_io()` must be NULL.

Returns: EOK Success.

Classification: QNX Neutrino, QNX 4

Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Caveats: To ensure that the correct physical address will be used, you must use the buffer allocated by `usbd_alloc()` for the `addr` parameter.

See also: `usbd_descriptor()`, `usbd_feature()`, `usbd_io()`, `usbd_setup_bulk()`, `usbd_setup_control()`, `usbd_setup_interrupt()`, `usbd_setup_isochronous()`, `usbd_status()`
**usbd_status()**

Get specific device status

**Synopsis:**

```c
#include <sys/usbd.h>

int usbd_status( struct usbd_device *device,
                 _uint16 rtype,
                 _uint16 index,
                 void *addr,
                 _uint32 len )
```

**Arguments:**

- `device` An opaque handle used to identify the USB device.
- `rtype` Type of request (e.g. `USB_RECIPIENT_DEVICE`, `USB_RECIPIENT_INTERFACE`, `USB_RECIPIENT_ENDPOINT`, `USB_RECIPIENT_OTHER`, `USB_TYPE_STANDARD`, `USB_TYPE_CLASS`, `USB_TYPE_VENDOR`).
- `index` This varies, depending on the request. It’s used for passing a parameter to the device.
- `addr` Address for start of transfer — you must use the buffer allocated by `usbd_alloc()`.
- `len` The length (in bytes) of the data transfer.

**Library:**

libusbd

**Description:**

You use the `usbd_status()` function to get specific device status.

**Returns:**

- **EOK** Success.
- **EMSGSIZE** Buffer too small for descriptor.
- **ENOMEM** No memory for URB.
- **ENODEV** Device was removed.

**Classification:**

QNX Neutrino, QNX 4
### usbd_status()

#### Safety

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_descriptors()`, `usbd_features()`, `usbd_ios()`, `usbd_setup_bulk()`,
`usbd_setup_control()`, `usbd_setup_interrupt()`, `usbd_setup_isochronous()`,
`usbd_setup_vendor()`
usbd_string()  
Get a string descriptor

Synopsis:

```c
#include <sys/usbd.h>

char *usbd_string( struct usbd_device *device,
                    _uint8 index,
                    int langid );
```

Arguments:

- `device`: An opaque handle used to identify the USB device.
- `index`: Index into the device’s (optional) string table.
- `langid`: Language ID. The `usbd_languages_descriptor()` function provides the supported `langids` for the device. If you specify 0, the `usbd_string()` function will select the first/only supported language.

Library:

- `libusbd`

Description:

The `usbd_string()` function lets you obtain a string from the USB device’s table of strings.

Typically, the string table may contain the names of the vendor, the product, etc. The string table is optional.

Note that the strings are actually in Unicode/wide characters, so `usb_string()` also conveniently converts them to UTF-8 (byte stream) for you.

Note that `usbd_string()` places the result string in a static buffer that’s reused every time the function is called.

Returns:

A pointer to the string in an internal static buffer, or NULL on error.

Classification:

- QNX Neutrino, QNX 4

Safety

- Cancellation point: No
- Interrupt handler: No

```
continued…
```
Safety

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>No</td>
</tr>
</tbody>
</table>

See also:

usbd_argsLookup(), usbd_configuration_descriptor(), usbd_device_lookup(),
usbd_device_extra(), usbd_device_descriptor(), usbd_endpoint_descriptor(),
usbd_hcd_info(), usbd_hub_descriptor(), usbd_interface_descriptor(),
usbd_languages_descriptor(), usbd_parse_descriptors(), usbd_urb_status()
## Synopsis:

```c
#include <sys/usbdi.h>

int usbd_topology( struct usbd_connection *connection,
                   usbd_bus_topology_t *tp )

int usbd_topology_ext( struct usbd_connection *connection,
                       _uint8 busno,
                       usbd_bus_topology_t *tp )
```

## Arguments:

- **connection**: An opaque handle that identifies the USB stack, obtained by calling `usbd_connect()`.
- **bus** (only for `usbd_topology_ext()`) The index of the bus that you want the topology for.
- **tp**: A pointer to a `usbd_bus_topology_t` data structure that this function fills in; see below.

## Library:

**libusbdi**

## Description:

You can use the `usbd_topology()` or `usbd_topology_ext()` function to get the USB bus physical topology.

For more information on USB bus topology, see sections 4.1.1 and 5.2.3 in the USB Specification v1.1.

If your system has more than one bus, you can call `usbd_topology_ext()` to get information about a specific one. The `usbd_topology()` function gets information about the first bus; calling it is the same as calling `usbd_topology()` with a `bus` argument of 0.

The `usbd_bus_topology_t` structure is defined as follows:

```c
typedef struct usbd_port_attachment {
  _uint8 upstream_devno;
  _uint8 upstream_port;
  _uint8 upstream_port_speed;
  _uint8 upstream_HC;
  _uint8 reserved[4];
} usbd_port_attachment_t;

typedef struct usbd_bus_topology {
  usbd_port_attachment_t ports[64];
} usbd_bus_topology_t;
```
The structure contains an array of `usb_port_attachment_t` structures, one per device. The `usb_port_attachment_t` structure contains at least the following:

- `upstream_devno` (The device number of the upstream hub (0 if it’s a root port).)
- `upstream_port` (The port number the device is connected to.)
- `upstream_port_speed` (The port speed that the device is operating at; one of the following:
  - 0 — full
  - 1 — low
  - 2 — high
- `upstream_HC` (The bus or host controller that the device is connected to.)

The `upstream_devno` field will contain a value other than `0xff` to indicate a valid attachment.

**Returns:**

- `EOK` Success.
- `ENODEV` The device was removed.

**Classification:**

QNX Neutrino, QNX 4

**Safety**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>Yes</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**See also:**

`usbd_connect()`
Synopsis:

```c
#include <sys/usbdi.h>

int usbd_urb_status( struct usbd_urb *urb,
                     _uint32 *status,
                     _uint32 *len )
```

Arguments:

- **urb**: An opaque handle (from `usbd_alloc_urb()`).
- **status**: Completion status (see below).
- **len**: The actual length (in bytes) of the data transfer.

Library:

`libusbdi`

Description:

You use the `usbd_urb_status()` function to extract completion status and data-transfer length from a URB.

Completion status

The **status** field contains the completion status information, which includes the following flags:

- **USBD_STATUS_INPROG**: The operation is in progress.
- **USBD_STATUS_CMP**: The operation is complete.
- **USBD_STATUS_CMP_ERR**: The operation is complete, but an error occurred.
- **USBD_STATUS_TIMEOUT**: The operation timed out.
- **USBD_STATUS_ABORTED**: The operation aborted.
- **USBD_STATUS_CRC_ERR**: The last packet from the endpoint contained a CRC error.
USBD_STATUS_BITSTUFFING
   The last packet from the endpoint contained a bit-stuffing violation.

USBD_STATUS_TOGGLE_MISMATCH
   The last packet from the endpoint had the wrong data-toggle PID.

USBD_STATUSSTALL
   The endpoint returned a STALL PID.

USBD_STATUS_DEV_NOANSWER
   Device didn’t respond to token (IN) or didn’t provide a handshake (OUT).

USBD_STATUS_PID_FAILURE
   Check bits on PID from endpoint failed on data PID (IN) or handshake (OUT).

USBD_STATUS_BAD_PID
   Receive PID was invalid or undefined.

USBD_STATUS_DATA_OVERRUN
   The endpoint returned more data than the allowable maximum.

USBD_STATUS_DATA_UNDERRUN
   The endpoint didn’t return enough data to fill the specified buffer.

USBD_STATUS_BUFFER_OVERRUN
   During an IN, the host controller received data from the endpoint faster than it could be written to system memory.

USBD_STATUS_BUFFER_UNDERRUN
   During an OUT, the host controller couldn’t retrieve data fast enough.

USBD_STATUS_NOT_ACCESSED
   Controller didn’t execute request.

**Returns:**

- EOK  Success.
- EBUSY  URB I/O still active.
- ETIMEDOUT  Timeout occurred.
- EINTR  Operation aborted/interrupted.
- ENODEV  Device removed.
- EIO  I/O error.
Classification:

QNX Neutrino, QNX 4

<table>
<thead>
<tr>
<th>Safety</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cancellation point</td>
<td>No</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>No</td>
</tr>
<tr>
<td>Signal handler</td>
<td>No</td>
</tr>
<tr>
<td>Thread</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also:

usbd_args_lookup(), usbd_configuration_descriptor(), usbd_device_lookup(),
usbd_device_extra(), usbd_device_descriptor(), usbd_endpoint_descriptor(),
usbd_hcd_info(), usbd_hub_descriptor(), usbd_interface_descriptor(),
usbd_languages_descriptor(), usbd_parse_descriptors(), usbd_string()
Index

A
arguments, getting command-line 22
assumptions ix

B
bulk data transfers 8, 70
bus topology, getting information about 84

C
callbacks 8, 24, 29
class drivers
  hub 7
  library for 7
  printers 11
  shared memory 7
  source code for ix
  starting 7
  supported devices 29
threads, protecting resources in 31
typical operations 8
command-line arguments, getting 22
configuration
descriptor, getting 26
displaying 11
functions 16
selecting 8, 67
connection functions 15
control transfers 8, 72
conventions
typographical xii

d
data buffers 7
data transfers See transfers
DDK library
  functions 15
getting information about 48
descriptors
  configuration 26
device 36, 61
endpoint 41, 59
getting and setting 8, 32
hub 50
interface 52
language 56
string 82
devi-hirun 4
devices
  attaching to 8, 23
  configuration
displaying 11
  selecting 8, 67
descriptors
getting 36
  parsing 61
detaching from 8, 34
detaching from 8, 34
elementary memory, getting a pointer to 38
features, controlling 43
frame number and length, getting 47
handle, mapping instance identifier to 39
hub descriptors, getting 50
interface, selecting 8, 68
pipe, getting for associated 63
resetting 65
status of, getting 80
string descriptors, getting 82
supported 3
devu-ehci.so 7, 11
devu-ohci.so 7, 11
devu-prn 11
devu-uhci.so 7, 11
drivers
command-line arguments, getting 22
language IDs, getting supported 56
USB stack
connecting to 8, 28
disconnecting from 40

E
endpoint_descriptor_t 41
endpoints
clearing a stall condition on 66
descriptors
getting 41
initializing pipe described by 8, 59
number, getting for a pipe 64
Enhanced Host Controller Interface (EHCI) 7
enumerator 7

H
host controllers
getting information about 48
types 7
hubs
class driver for included in stack 7
descriptors, getting 50
supported 3

I
I/O functions 15
Input 4
insertion/removal 8, 24, 29
interfaces
descriptors, getting 52
functions 16
selecting 8, 68
interrupt transfers 8, 74
io-net 7
io-usb 11
isochronous transfers 8, 76
not currently supported 3

K
keyboards
controller, don’t reset 4
supported 3

L
language IDs, getting supported 56
library
about 7
functions 15
getting information about 48
libusbdi 15
limitations 3
looping, as alternate method of attaching 24
M
memory
  data transfers
    allocating 19
    freeing 45
    getting physical address of 58
  management functions 15
mice, supported 3
mutexes 31

O
Open Host Controller Interface (OHCI) 7

P
pathname delimiter in QNX Momentics documentation xiii
Photon 4
pipes
  closing 25
  endpoint number, getting 64
  getting associated device 63
  initializing 8, 59
  management functions 16
  not a UNIX term in this doc 8
  requests, aborting all 18
  resetting 66
printers
  class driver for 11
  supported 3

R
request blocks  See URBs (USB Request Blocks)

S
server 11

shared memory 7
split isochronous transfers, not currently supported 3
stack
  about 7
  drivers
    connecting to 8, 28
    disconnecting from 40
  shared memory 7
URBs (USB Request Blocks),
  submitting 8, 54
string descriptors, getting 82
system requirements 3

T
threads, protecting resources in 31
transfers
  bulk data 8, 70
  control 8, 72
  initiating 8, 54
  interrupt 8, 74
  isochronous 8, 76
  vendor-specific 8, 78
typographical conventions xii

U
Universal Host Controller Interface (UHCI) 7
URB_DIR_IN 70, 72, 74, 76, 78
URB_DIR_NONE 70, 72, 74, 76, 78
URB_DIR_OUT 70, 72, 74, 76, 78
URB_ISOCH_ASAP 76
URB_SHORT_XFER_OK 70, 72, 74, 76, 78
URBs (USB Request Blocks)
  allocating 21
  freeing 46
  getting status of 86
  setting up
    bulk data transfers 8, 70
    control transfers 8, 72
    interrupt transfers 8, 74
    isochronous transfers 8, 76
Index

vendor-specific transfers 8, 78
submitting 8, 54

usb 11
USB

descriptors, getting and setting 8, 32
link to www.usb.org ix
server 11
Specification revision 2.0 ix
USB_DESC_CONFIGURATION 32
USB_DESC_DEVICE 32
USB_DESC_HUB 32
USB_DESC_STRING 32
usb_port_attachment_t 85
USB_RECIPIENT_DEVICE 32, 43, 72, 78, 80
USB_RECIPIENT_ENDPOINT 32, 43, 72, 80
USB_RECIPIENT_INTERFACE 32, 43, 72, 80
USB_RECIPIENT_OTHER 32, 43, 72, 80
USB_TYPE_CLASS 32, 43, 72, 80
USB_TYPE_STANDARD 32, 43, 72, 80
USB_TYPE_VENDOR 32, 43, 72, 80
USB_VERSION 28
usbd_abort_pipe() 18
usbd_alloc_urb() 21
usbd_alloc() 19
usbd_args_lookup() 22
usbd_attach() 8, 23
usbd_bus_topology_t 84
usbd_close_pipe() 25
usbd_configuration_descriptor_t 26
usbd_configuration_descriptor() 26
usbd_connect_parm_t 28
USBD_CONNECT_WAIT 29
USBD_CONNECT_WILDCARD 29
usbd_connect() 8, 28
usbd_desc_node 61
usbd_descriptor() 8, 32
usbd_detach() 8, 34
usbd_device 63
usbd_device_descriptor_t 36
usbd_device_descriptor() 36
usbd_device_extra() 38
usbd_device_ident_t 29
usbd_device_instance_t 23
usbd_device_lookup() 39

usbd_disconnect() 40
usbd_endpoint_descriptor() 41
usbd_feature() 43
usbd_free_urb() 46
usbd_free() 45
usbd_funcs_t 29
usbd_get_frame() 47
usbd_hcd_ext_info(),usbd_hcd_info() 48
usbd_hcd_info_t 48
usbd_hub_descriptor_t 50
usbd_hub_descriptor() 50
usbd_interface_descriptor_t 52
usbd_interface_descriptor() 52
usbd_ioc() 8, 54
usbd_languages_descriptor() 56
usbd_mphys() 58
usbd_open_pipe() 8, 59
usbd_parse_descriptors() 61
usbd_pipe_device() 63
usbd_pipe_endpoint() 64
usbd_reset_device() 65
usbd_reset_pipe() 66
usbd_select_config() 8, 67
usbd_select_interface() 8, 68
usbd_setup_bulk() 8, 70
usbd_setup_control() 8, 72
usbd_setup_interrupt() 8, 74
usbd_setup_isochronous() 8, 76
usbd_setup_vendor() 8, 78
USBD_STATUS_ABORTED 86
USBD_STATUS_BAD_PID 87
USBD_STATUS_BITSTUFFING 87
USBD_STATUS_BUFFER_OVERRUN 87
USBD_STATUS_BUFFER_UNDERRUN 87
USBD_STATUS_CMP 86
USBD_STATUS_CMP_ERR 86
USBD_STATUS_CRC_ERR 86
USBD_STATUS_DATA_OVERRUN 87
USBD_STATUS_DATA_UNDERRUN 87
USBD_STATUS_DEV_NOANSWER 87
USBD_STATUS_INPROG 86
USBD_STATUS_NOT_ACCESSSED 87
USBD_STATUS_PID_FAILURE 87
USBD_STATUSSTALL 87
USBD_STATUS_TIMEOUT 86
USBD_STATUS_TOGGLE_MISMATCH 87
**Index**

- `usbd_status() 80`
- `usbd_string_descriptor_t 56`
- `usbd_string() 82`
- `USBD_TIME_DEFAULT 54`
- `USBD_TIME_INFINITY 54`
- `usbd_topoogy(),usbd_topoogy_ext() 84`
- `usbd_urb_status() 86`
- `USBD_VERSION 28`
- `utilities 11`

**V**

- `vendor-specific transfers 8, 78`