Freescale MQX™
USB Host API Reference Manual

Document Number: MQXUSBHOSTAPIRM
Rev. 1
04/2009

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 26668334
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

freescale™
semiconductor

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. ARC, the ARC logo, ARCangel, ARCform, ARChitect, ARCompact, ARCtangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, MetaDeveloper, MQX, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/RTCS, RTCS, SeeCode, TotalCore, Turbo186, Turbo86, V8 μ RISC, V8 microRISC, and VAutomation are trademarks of ARC International. High C and MetaWare are registered under ARC International.
All other product or service names are the property of their respective owners.

© 1994-2008 ARC™ International. All rights reserved.

© Freescale Semiconductor, Inc. 2009. All rights reserved.

Document Number: MQXUSBHOSTAPIRM
Rev. 0
4/2009

Chapter 1
Before you begin

Chapter 2

USB Host API Overview

Chapter 3

Host API Functions

Chapter 4

Device Framework Functions

Chapter 5

Data Types

Chapter 1 Before you begin

1.1 About this Book

This *USB Host API Reference Manual* describes the following products:

- USB 1.1 Host API
- USB 2.0 Host API

This book does not distinguish between USB 1.1 and USB 2.0 information unless there is a difference between the two.

This book contains the following topics:

- Chapter 1 — "Before you begin"
- Chapter 2 — "USB Host API overview"
- Chapter 3 — "Host API Functions"
- Chapter 4 — "Device framework functions"
- Chapter 5 — "Data types"

1.2 Where to Go for More Information

We recommend that you consult the following reference material:

- Universal Serial Bus Specification Revision 1.1
- Universal Serial Bus Specification Revision 2.0
- For more information, see `www.usb.org`

1.3 Document Conventions

- Notes — Notes point out important information.

Note

Names of command-line options are case-sensitive.

- Cautions — Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION

Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **`#if`** or **`#end`**, C comment delimiters, or invalid C tokens.

Chapter 2
USB Host API Overview

2.1 USB Host at a Glance

The USB Host provides USB Device drivers and applications with a uniform view of the I/O system. Since the USB Host manages the attachment and detachment of peripherals along with their power requirements dynamically, all hardware implementation details can be hidden from applications. The USB Host determines which device driver to load for the connected device, and assigns a unique address to the device for run-time data transfers. The USB Host also manages data transfers and bus bandwidth allocation.

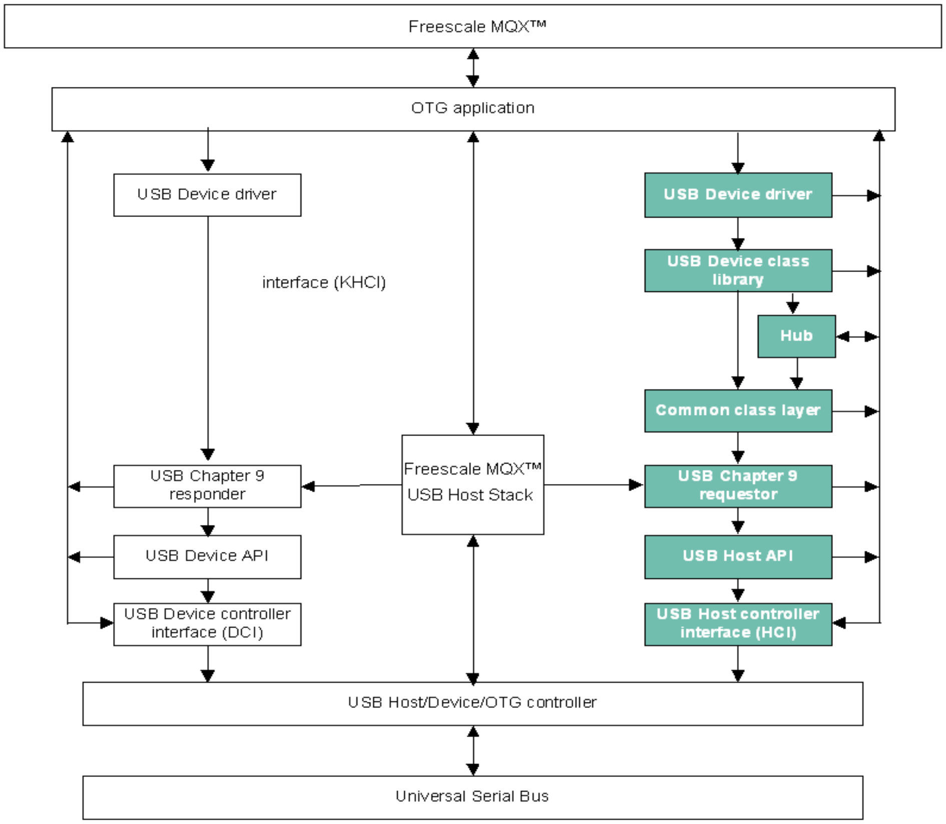
The Freescale MQX™ USB Host stack includes the following components:

- USB Device class library
- USB Host API—a hardware-independent application interface
- USB Host controller interface (KHCI)—low-level functions that are called by the USB Host API to interact with USB Host controller hardware

2.2 Interaction between the USB Host and USB Devices

In a USB system, the USB Host initiates all data transfers and configures all devices that are attached to it directly or indirectly through a connected USB hub. All USB Devices are slaves that must only respond to requests from the USB Host.

USB Devices send and receive data to/from the USB Host using a standard USB format. USB 1.1 peripherals can operate at 12 Mbps or 1.5 Mbps, while targets of up to 480 Mbps can be achieved by USB 2.0 Devices. Both USB 1.1 and 2.0 Devices can interoperate in a USB 2.0 system—a USB 2.0 Host can detect the capabilities of each type of device and negotiate transmission speeds on a device-by-device basis.



2.3 Using the Freescale MQX™ USB Host API

To use the Freescale MQX™ USB Host API, follow these general steps. Each API functions are described in next chapters.

1. Initialize the USB Host controller interface (**_usb_host_init()**).
2. Optionally register services for types of events (**_usb_host_register_service()**).

NOTE

> Before transferring any packets, the application should determine that the enumeration process has been completed. This can be done by registering a callback function that notifies the application when the enumeration has been completed.

3. Open the pipe for a connected device or devices (**_usb_host_open_pipe()**).
4. Send control packets to configure the device or devices (**_usb_host_send_setup()**).
5. Send (**_usb_host_send_data()**) and receive (**_usb_host_recv_data()**) data on pipes.
6. If required, cancel a transfer on a pipe (**_usb_host_cancel_transfer()**).
7. If applicable, unregister services for pipes or types of events (**_usb_host_unregister_service()**) and close pipes for disconnected devices (**_usb_host_close_pipe()**).

8. Shut down the USB Host controller interface (**_usb_host_shutdown()**).

Alternatively:

1. Define the table of driver capabilities that the application uses (as follows):

Example 2-1. Sample driver info table

```
static  USB_HOST_DRIVER_INFO DriverInfoTable[ ] =
{
   {
      /* Vendor ID per USB-IF */
      {0x00,0x00},

      /* Product ID per manufacturer */
      {0x00,0x00},

      /* Class code */
      USB_CLASS_MASS_STORAGE,

      /* Sub-Class code */
      USB_SUBCLASS_MASS_UFI,

      /* Protocol */
      USB_PROTOCOL_MASS_BULK,

      /* Reserved */
      0,

      /* Application call back function */
      usb_host_mass_device_event

   },
   {

      /* Vendor ID per USB-IF  */
      {0x00,0x00},

      /* Product ID per manufacturer */
      {0x00,0x00},

      /* Class code */
      USB_CLASS_PRINTER,

      /* Sub-Class code */
      USB_SUBCLASS_PRINTER,

      /* Protocol */
      USB_PROTOCOL_PRT_BIDIR,

      /* Reserved */
      0,

      /* Application call back function */
      usb_host_prt_device_event

   },
   {
```

```
  /* All-zero entry terminates */
  {0x00,0x00},

  /* driver info list. */
  {0x00,0x00},
   0,
   0,
   0,
   0,
   NULL
   }
};
```

2. Initialize the USB Host controller interface (**_usb_host_init**()).
3. The application should then register this table with the host stack by calling the **usb_host_driver_info_register host** API function.
4. Optionally register services for types of events (**_usb_host_register_service**()).
5. Wait for the callback function (specified in the `driverinfo` table) to be called.
6. Check for the events in the callback function: One of `ATTACH`, `DETACH`, `CONFIG` or `INTF`.
   `ATTACH`: indicates a newly attached device was just enumerated and a default configuration was selected
   `DETACH`: the device was detached
   `CONFIG`: A new configuration was selected on the device
   `INTF`: A new interface was selected on the device.
7. If it is an attach event, then select an interface by calling the host API function **usb_hostdev_select_interface**.
8. After the `INTF` event is notified in the callback function, issue class-specific commands by using the class API.
9. Open the pipe for a connected device or devices (**_usb_host_open_pipe**()).
10. Get the pipe handle by calling the host API function **_usb_hostdev_find_pipe_handle**.
11. Transfer data by using the host API functions **_usb_host_send_data** and/or **_usb_host_recv_data**.
12. If required, cancel a transfer on a pipe (**_usb_host_cancel_transfer**()).
13. If applicable, unregister services for types of events (**_usb_host_unregister_service**()) and close pipes for disconnected devices (**_usb_host_close_pipe**()).
14. Shut down the USB Host controller interface (**_usb_host_shutdown**()).

2.4 Transaction Scheduling

For USB 1.1, transaction scheduling is managed by USB Host API. For USB 2.0, USB Host API manages the bandwidth allocation and enqueing the transfers. The enqueued transfer is then managed by the hardware.

If using USB 2.0 hardware, the KHCI determines and allocates the required bandwidth over the whole frame list when **_usb_host_open_pipe()** is called (the size of the frame list is determined from the parameter passed to **_usb_host_init()**. The pipe can then be used to queue a transfer (by calling **_usb_host_send_data()** and **_usb_host_recv_data()**) that is scheduled every INTERVAL units of time (the value is defined in PIPE_INIT_PARAM_STRUCT). When the host is the data source, an application should provide timely data by calling **_usb_host_send_data()**. When the application determines that the transfer has been completed, it should relinquish the allocated bandwidth if the bandwidth is not required further. This can be done by calling **_usb_host_close_pipe()**.

Interrupt data transfers—provides the reliable, limited-latency delivery of data. If using USB 2.0 hardware, the KHCI determines and allocates the required bandwidth over the whole frame list when **_usb_host_open_pipe()** is called (size of frame list is determined from the parameter passed to **_usb_host_init()**. The pipe can then be used to queue a transfer (by calling **_usb_host_send_data()** and **_usb_host_recv_data()**) that is scheduled every INTERVAL units of time (the value is defined in PIPE_INIT_PARAM_STRUCT). For USB1.1, the interval is in milliseconds. For USB 2.0, it is in terms of 125-microsecond units. The NAK_COUNT field in PIPE_INIT_PARAM_STRUCT is ignored for interrupt data transfers.

Control data transfers—to configure devices when they are first attached and control pipes on a device.

Bulk data transfers—for large amounts of data that can be delivered in sequential bursts.

Within pipes opened for the same type of data, scheduling is round robin, even if the packet is NAKed; that is, the transaction has to be retried when bus time is available.

Control and bulk data transfers—for USB 1.1, after NAK_COUNT NAK responses per frame, the transaction is deferred to the next frame. For USB 2.0, the host controller does not execute a transaction if NAK_COUNT NAK responses are received on the pipe

2.5 USB Host API Summary

Table 2-1” summarizes the USB Host API functions.

Table 2-1. Summary of USB Host API

_usb_host_bus_control	Control the operation of the bus
_usb_host_cancel_transfer	Cancel a specific transfer on a pipe
_usb_host_close_all_pipes	Close all pipes
_usb_host_close_pipe	Close a pipe
_usb_host_driver_info_register	Register driver information
_usb_host_get_frame_number	Get the current frame number
_usb_host_get_micro_frame_number	Get the current microframe number
_usb_host_get_transfer_status	Get the status of a specific transfer on a pipe
_usb_host_init	Initialize the USB Host controller interface
_usb_host_open_pipe	Open the pipe between a host and a device endpoint

Table 2-1. Summary of USB Host API (continued)

_usb_host_recv_data	Receive data on a pipe
_usb_host_register_service	Register a service for a pipe or specific event
_usb_host_send_data	Send data on a pipe
_usb_host_send_setup	Send a setup packet on a control pipe
_usb_host_shutdown	Shut down the USB Host controller interface
_usb_host_unregister_service	Unregister a service for a pipe or specific event
_usb_hostdev_find_pipe_handle	Find a pipe for the specified interface
_usb_hostdev_get_buffer	Get a buffer for a particular device operation
_usb_hostdev_get_descriptor	Get the specified USB descriptor that exists in device specific data structure
_usb_hostdev_select_config	Select a new configuration of the device
_usb_hostdev_select_interface	Select a new interface on the device

Chapter 3
Host API Functions

3.1 How to Read Prototype Definitions

3.1.1 example_function

A short description of what **example_function()** does.

Synopsis

```
<return_type> example_function(
  <type_1>  parameter_1,
  ...
  <type_n>  parameter_n)
```

Parameters

parameter_1 [in], [out], [in/out] — Short description of *parameter_1*

Returns

- Return value (success)
- Return value (failure)

Traits

Any of the following that might apply for the function:

- it blocks or the conditions under which it might block
- it must be started as a task
- it creates a task
- pre-conditions that might not be obvious
- any other restrictions or special behavior

See also

- For functions that are listed, see the descriptions in this chapter
- For data types that are listed, see the descriptions in Chapter 5 “Data Types,” on page 51”

Description — Any pertinent information that is not specified in the preceding table or short description is included here.

3.1.2 _usb_host_bus_control

Control the operation of the bus.

Synopsis

```
void  _usb_host_bus_control(
  usb_host_handle       hci_handle,
  uint_8                bus_control)
```

Parameters

hci_handle [in] —USB Host controller handle

bus_control [in] —Operation to be performed on the bus; one of:

USB_ASSERT_BUS_RESET—reset the bus

USB_ASSERT_RESUME—if the bus is suspended, resume operation

USB_DEASSERT_BUS_RESET—
bring the bus out of reset mode

USB_DEASSERT_RESUME—bring the bus out of resume mode

USB_NO_OPERATION—make the bus idle

USB_RESUME_SOF—generate and transmit start-of-frame tokens

USB_SUSPEND_SOF—do not generate start-of-frame tokens

Returns

Traits

See also

Description — The function controls the bus operations such as asserting and deasserting the bus reset, asserting and deasserting resume signalling, suspending and resuming the SOF generation.

3.1.3 _usb_host_cancel_transfer

Cancel the specified transfer on the pipe.

Synopsis

```
uint_32  _usb_host_cancel_transfer(
  _usb_host_handle   hci_handle,
  _usb_pipe_handle   pipe_handle,
  uint_32            transfer_number)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_handle [in] — Pipe handle

transfer_number [in] — Specific transfer to cancel

Should correspond the TR_INDEX field in the transfer request (PIPE_INIT_PARAM_STRUCT) for the particular transfer when **_usb_host_send_setup()**, **_usb_host_send_data()**, or **_usb_host_recv_data()** was called.

Returns

Status of the transfer prior to cancellation (see **_usb_host_get_transfer_status()**) (success)

USBERR_INVALID_PIPE_HANDLE — Valid for USB 2.0 Host API only (failure; pipe_handle is not valid)

Traits

See also

_usb_host_get_transfer_status(), **_usb_host_recv_data**(), **_usb_host_send_data**(), **_usb_host_send_setup**(), TR_INIT_PARAM_STRUCT

Description — The function cancels the specified transfer on the pipe at the hardware level. It will then call the callback function for that transaction (if there was one registered for that transfer by using the TR_INIT_PARAM_STRUCT) with the status value as **USBERR_SHUTDOWN** indicating that the transfer was cancelled.

3.1.4 _usb_host_close_all_pipes

Close all pipes.

Synopsis

```
void  _usb_host_close_all_pipes(
  _usb_host_handle_   hci_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

Returns

Traits

See also

_usb_host_close_pipe(), **_usb_host_open_pipe**()

Description — The function removes all pipes from the list of open pipes.

3.1.5 _usb_host_close_pipe

Close the specified pipe functions.

Synopsis

```
uint_32  _usb_host_close_pipe(
  _usb_host_handle_  hci_handle,
  _usb_pipe_handle   pipe_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_handle [in] — Pipe handle

Returns

USB_OK (success)

USBERR_INVALID_PIPE_HANDLE (failure; *pipe_handle* is not valid)

Traits

See also

_usb_host_close_all_pipes(), **_usb_host_open_pipe**()

Description — The function removes the pipe from the list of open pipes.

3.1.6 _usb_host_driver_info_register

Register driver information

Synopsis

```
USB_STATUS _usb_host_driver_info_register(
  _usb_host_handle  host_handle,
  pointer           info_table_ptr)
```

Parameters

host_handle [in] — USB host

info_table_ptr [in] — Device info table

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

USB_HOST_DRIVER_INFO

Description — This function is used by the application to register a driver for a device with a particular vendor ID, product ID, class, subclass and protocol code.

3.1.7 _usb_host_get_frame_number

Get the current frame number — for USB 2.0 Host API only.

Synopsis

```
uint_32  _usb_host_get_frame_number(
  _usb_host_handle  hci_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

Returns

Current frame number

Traits

See also

_usb_host_get_micro_frame_number()

Description — An application can use the function to determine at which frame number a particular transaction should be scheduled.

3.1.8 _usb_host_get_micro_frame_number

Get the current microframe number — for USB 2.0 Host API only.

Synopsis

```
uint_32  _usb_host_get_micro_frame_number(
  _usb_host_handle   hci_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

Returns

Current microframe number

Traits

See also

_usb_host_get_frame_number ()

Description — An application can use the function to determine at which microframe number a particular transaction should be scheduled.

3.1.9 _usb_host_get_transfer_status

Get the status of the specified transfer on the pipe.

Synopsis

```
uint_32  _usb_host_get_transfer_status(
 usb_pipe_handle  pipe_handle,
 uint_32          transfer_number)
```

Parameters

pipe_handle [in] — Pipe handle

transfer_number [in] — Specific transfer number on the pipe

Should correspond the **TR_INDEX** field in the transfer request (TR_INIT_PARAM_STRUCT) for the particular transfer when **_usb_host_send_setup()**, **_usb_host_send_data()**, or **_usb_host_recv_data()** was called.

Returns

Status of the transfer; one of:

- **USB_STATUS_IDLE** (no transfer is queued or completed)
- **USB_STATUS_TRANSFER_QUEUED** (transfer is queued, but is not in progress)
- **USB_STATUS_TRANSFER_IN_PROGRESS** (transfer is queued in the hardware and is in progress)

or

- **USBERR_INVALID_PIPE_HANDLE** (error; *pipe_handle* is not valid)

Traits

Blocks

See also

_usb_host_cancel_transfer(), _usb_host_get_transfer_status(), _usb_host_recv_data(), _usb_host_send_data(), _usb_host_send_setup(), TR_INIT_PARAM_STRUCT

Description — The function gets the status of the specified transfer on the specified pipe. It reads the status of the transfer.

To determine whether a receive or send request has been completed, the application can call **_usb_host_get_transfer_status**() to check whether the status is **USB_STATUS_IDLE**.

3.1.10 _usb_host_init

Initialize the USB Host controller interface data structures and the controller interface.

Synopsis

```
uint_32  _usb_host_init(
  uint_8                  devnum,
  uint_32                 frame_list_size,
  _usb_host_handle _PTR_  hci_handle)
```

Parameters

devnum [in] — Device number of the USB Host controller to initialize

frame_list_size [in] — Number of elements in the periodic frame list; one of:

256

512

1024 (default)

(ignored for USB 1.1)

hci_handle [out] — Pointer to a USB Host controller handle

Returns

USB_OK (success)

Error code (failure; see errors)

Traits

See also

_usb_host_shutdown()

Description — The function calls a KHCI function to initialize the USB Host hardware and install an ISR that services all interrupt sources on the USB Host hardware.

The function also allocates and initializes all internal host-specific data structures and USB Host internal data and returns a USB Host controller handle for subsequent use with other USB Host API functions.

If *frame_list_size* is not a valid value, 1024 is assumed and **USB_OK** is returned.

Errors

USBERR_ALLOC

Failed to allocate memory for internal data structures.

USBERR_DRIVER_NOT_INSTALLED

Driver for the host controller is not installed (reported only when using USB Host API with the Freescale MQX™ RTOS).

USBERR_INSTALL_ISR

Could not install the ISR (reported only when using USB Host API with the MQX RTOS).

3.1.11 _usb_host_open_pipe

Open a pipe between the host and the device endpoint.

Synopsis

```
uint_32  _usb_host_open_pipe(
  _usb_host_handle            hci_handle,
  PIPE_INIT_PARAM_STRUCT_PTR  pipe_init_params_ptr,
  _usb_pipe_handle _PTR_      pipe_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_init_params_ptr [in] — Pointer to the pipe initialization parameters

pipe_handle [out] — Pipe handle

Returns

Pipe handle (success)

Error code (failure: see errors)

Traits

See also

_usb_host_close_all_pipes(), _usb_host_close_pipe(), PIPE_INIT_PARAM_STRUCT

Description — The function initializes a new pipe for the specified USB device address and endpoint and returns a pipe handle for subsequent use with other USB Host API functions.

All bandwidth allocation for a pipe is done when this function is called. If the services of a pipe are not required or the bandwidth requirements change, the pipe should be closed.

Errors

USBERR_BANDWIDTH_ALLOC_FAILED

Required bandwidth could not be allocated (valid for USB 2.0 stack only).

USBERR_OPEN_PIPE_FAILED

failure; *open_pipe* failed

3.1.12 _usb_host_recv_data

Receive data on a pipe.

Synopsis

```
uint_32  _usb_host_recv_data(
  _usb_host_handle              hci_handle,
  _usb_pipe_handle              pipe_handle,
   TR_INIT_PARAM_STRUCT_PTR    tr_params_ptr)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_handle [in] — Pipe handle

tr_ptr [in] — Pointer to the transfer request parameters

Returns

USB_STATUS_TRANSFER_QUEUED (success)

Error code (failure; see errors)

Traits

Does not block

See also

_usb_host_get_transfer_status(), _usb_host_open_pipe(), _usb_host_send_data().
PIPE_INIT_PARAM_STRUCT, TR_INIT_PARAM_STRUCT

Description — The function calls a KHCI function to queue the receive request and then returns. Multiple receive requests on the same endpoint can be queued.

The receive transfer completes when the host receives exactly RX_LENGTH bytes (defined in TR_INIT_PARAM_STRUCT) on the specified pipe, or the last packet received on the pipe is less than MAX_PACKET_SIZE (set through PIPE_INIT_PARAM_STRUCT and calling **_usb_host_open_pipe()**). For USB 1.1, if RX_LENGTH is greater than MAX_PACKET_SIZE, the transfer is set to MAX_PACKET_SIZE bytes.

To check whether a transfer has been completed, the application can either:

- call **_usb_host_get_transfer_status()** and confirm a return status of **USB_STATUS_IDLE**
- provide a callback function (with parameters for length and transfer number) that can be used to notify the application that the transfer has been completed (see **_usb_host_open_pipe()**).

For information on how transactions are scheduled, see "Transaction Scheduling" on page 10.

Errors

USBERR_INVALID_PIPE_HANDLE

pipe_handle is not valid.

USB_STATUS_TRANSFER_IN_PROGRESS

A previously queued transfer on the pipe is still in progress, and the pipe cannot accept any more transfers until the previous one has been completed.

3.1.13 _usb_host_register_service

Register a service for a specific event.

Synopsis

```
uint_32  _usb_host_register_service(
  _usb_host_handle  hci_handle,
  uint_8            type,
  void (_CODE_PTR_  service)(pointer  callbk_ptr,
                             uint_32  event_param)
```

Parameters

hci_handle [in] — USB Host controller handle

type [in] — Event to service; one of:

USB_SERVICE_ATTACH—device has been connected to the bus

USB_SERVICE_DETACH—device has been disconnected from the bus

USB_SERVICE_HOST_RESUME—resume the host

USB_SERVICE_SYSTEM_ERROR—system error occurred while processing USB requests

service [in] — Pointer to the callback function

callbk_ptr [in] — Pointer to a USB Host controller handle

event_param [in] — Event-specific parameter

Returns

USB_OK (success)

Error code (failure; see errors)

Traits

See also

_usb_host_unregister_service()

Description — The function initializes a linked list of data structures with *event* and registers the callback function to service that event.

When the specific event (such as a device attach event) occurs, required information is collected as *event_param*, and *service* is called with *event_param* as a parameter.

Errors

USBERR_ALLOC

Failed to allocate memory for internal data structure.

USBERR_OPEN_SERVICE

Service was already registered.

3.1.14 _usb_host_send_data

Send data on a pipe.

Synopsis

```
uint_32  _usb_host_send_data(
  _usb_host_handle         hci_handle,
  _usb_pipe_handle         pipe_handle,
  TR_INIT_PARAM_STRUCT_PTR  tr_params_ptr)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_handle [in] — Pipe handle

tr_ptr [in] — Pointer to the transfer request

Returns

USB_STATUS_TRANSFER_QUEUED (success)

Error code (failure; see errors)

Traits

Does not block

See also

_usb_host_get_transfer_status(), **_usb_host_recv_data**()

, **PIPE_INIT_PARAM_STRUCT**, **TR_INIT_PARAM_STRUCT**

Description — The function calls a KHCI function to queue the send request and then returns. Multiple send requests on the same endpoint can be queued.

The send transfer completes when the host transmits exactly TX_LENGTH bytes (defined in **TR_INIT_PARAM_STRUCT**) on the specified pipe, or the last packet transmitted on the pipe is less than MAX_PACKET_SIZE (set through **PIPE_INIT_PARAM_STRUCT** and calling **_usb_host_open_pipe()**). For USB 1.1, for isochronous pipes, if TX_LENGTH is greater than MAX_PACKET_SIZE, the transfer is set to MAX_PACKET_SIZE bytes.

For USB 1.1, the data is broken up into packets before it is sent. If the transfer is for an integer multiple of MAX_PACKET_SIZE bytes, a zero-length packet is sent after the actual data. For example, if MAX_PACKET_SIZE is 16 and the transfer is for 36 bytes, the following size packets are sent: 16, 16, 4. However, if the transfer is for 32 bytes, the following size packets are sent: 16, 16, 0.

For USB 2.0, the hardware manages dividing the transfer into packets.

To check whether a transfer has been completed, the application can either:

- call **_usb_host_get_transfer_status()** and confirm a return status of **USB_STATUS_IDLE**
- provide a callback function with a length and transfer number parameter that can be used to notify the application that the transfer has been completed (see **TR_INIT_PARAM_STRUCT**)

Errors

USBERR_INVALID_PIPE_HANDLE

pipe_handle is not valid.

USB_STATUS_TRANSFER_IN_PROGRESS

A previously queued transfer on the pipe is still in progress and the pipe cannot accept any more transfers until the previous one has been completed.

3.1.15 _usb_host_send_setup

Send a setup packet on a control pipe.functions.

Synopsis

```
uint_32  _usb_host_send_setup(
  _usb_host_handle              hci_handle,
  _usb_pipe_handle              pipe_handle,
  TR_INIT_PARAM_STRUCT_PTR      tr_params_ptr)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_handle [in] — Pipe handle

tr_ptr [in] — Pointer to the transfer request

Returns

USB_STATUS_TRANSFER_QUEUED (success)

USB_STATUS_TRANSFER_IN_PROGRESS (failure; a previously queued transfer is still in progress)

USBERR_INVALID_PIPE_HANDLE (failure; *pipe_handle* is not valid)

Traits

See also

_usb_host_get_transfer_status(), **TR_INIT_PARAM_STRUCT**

Description — The function calls a KHCI function to queue the transfer and then returns. Once a control transfer request is queued, the KHCI manages or queues all phases of a control transfer.

NOTE

Before the application calls **_usb_host_send_setup()**, the control pipe must be idle: to determine whether the control pipe is idle, call **_usb_host_get_transfer_status()** and confirm a return status of **USB_STATUS_IDLE**.

3.1.16 _usb_host_shutdown

Shut down the USB Host controller interface.

Synopsis

```
     void  _usb_host_shutdown(
       _usb_host_handle  hci_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

Returns

Traits

See also

_usb_host_init()

Description — The function calls a KHCI function to stop the specified USB Host controller. Call the function when the services of the USB Host controller are no longer required, or if the USB Host controller needs to be reconfigured.

The function additionally does the following:

1. terminates all transfers
2. unregisters all services
3. disconnects the host from the USB bus
4. frees all memory that the USB Host allocated for its internal data

3.1.17 _usb_host_unregister_service

Unregister a service for a type of event.

Synopsis

```
uint_32 _usb_host_unregister_service(
  _usb_host_handle   hci_handle,
  uint_8             event)
```

Parameters

hci_handle [in] — USB Host controller handle

event [in] — Service to unregister (see **_usb_host_register_service()**)

Returns

USB_OK (success)

USBERR_CLOSED_SERVICE (failure: the specified service was not previously registered)

Traits

See also

_usb_host_register_service()

Description — The function unregisters the callback function that services the event As a result, the event can no longer be serviced by a callback function.

3.1.18 _usb_hostdev_find_pipe_handle

Find a specific pipe for the specified interface.

Synopsis

```
_usb_pipe_handle _usb_hostdev_find_pipe_handle(
    _usb_device_instance_handle      dev_handle,
    _usb_device_descriptor_handle    intf_handle,
   _uint_8                           pipe_type,
     _uint_8                                                      pipe_direction)
```

Parameters

dev_handle [in] — USB device

intf_handle [in] — Interface handle

pipe_type [in] — Pipe type; one of:

USB_ISOCHRONOUS_PIPE

USB_INTERRUPT_PIPE

USB_CONTROL_PIPE

USB_BULK_PIPE

pipe_direction [in] — Pipe direction (ignored for control pipe); one of:

USB_RECV

USB_SEND

Returns

Pipe handle (success)

NULL

Traits

See also

_usb_hostdev_select_interface

Description — Function to find a pipe with specified type and direction on the specified device interface. If the specified interface does not exist or is not selected by calling **_usb_hostdev_select_interface** then NULL is returned.

3.1.19 _usb_hostdev_get_buffer

Get a buffer for the device operation.

Synopsis

```
USB_STATUS _usb_hostdev_get_buffer(
  _usb_device_instance_handle  dev_handle,
  uint_32                      buffer_size,
  uchar_ptr _PTR_              buff_ptr)
```

Parameters

dev_handle [in] — USB device

buffer size [in] — Buffer size to get

buff_ptr [out] — Pointer to the buffer

Returns

Pointer to the buffer (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

Description — Applications should use this function to get buffers and other work areas that stay allocated until the device is detached. When the device is detached, these are all freed by the host system software.

3.1.20 _usb_hostdev_get_descriptor

Get a descriptor.

Synopsis

```
USB_STATUS _usb_hostdev_get_descriptor(
  _usb_device_instance_handle  dev_handle,
  descriptor_type              desc_type,
  uint8                        desc_index,
  uint8                        intf_alt,
  pointer_PTR_descriptor       _PTR_descriptor)
```

Parameters

dev_handle [in] — USB device

desc_type [in] — The type of descriptor to get

desc_index [in] — The descriptor index

intf_alt [in] — The interface alternate

pointer_PTR_descriptor [out] — Handle of the descriptor

Returns

handle of the descriptor (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

Description — When the host detects a newly attached device, the host system software reads the device and configuration (which includes interface and endpoint descriptors) descriptors and stores them in the internal device-specific memory. The application can request these descriptors by calling this function instead of issuing a device framework function request to get the descriptor from the device.

3.1.21 _usb_hostdev_select_config

Select the specified configuration for the device.

Synopsis

```
USB_STATUS _usb_hostdev_select_config(
  _usb_device_instance_handle  dev_handle,
  uint8                        config_no)
```

Parameters

dev_handle [in] — USB device

config_no [in] — Configuration number

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

_usb_host_ch9_get_configuration

Description — This function is used to select a particular configuration on the device. If the host had previously selected a configuration for the device then it will delete that configuration and select the new one. The host system sends a device framework command (_usb_host_ch9_get_configuration) to the device and then and then initializes and saves the configuration specific information in its internal data structures.

3.1.22 _usb_hostdev_select_interface

Select a new interface on the device.

Synopsis

```
USB_STATUS _usb_hostdev_select_interface(
  _usb_device_instance_handle        dev_handle,
  _usb_interface_descriptor_handle    intf_handle,
  pointer                            class_intf_ptr)
```

Parameters

dev_handle [in] — USB device

intf_handle [in] — Interface to be selected

class_intf_ptr [out] — Initialized class-specific interface struct

Returns

USB_OK and class-interface handle (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

_usb_host_ch9_set_interface

Description — This function should be used to select an interface on the device. It will delete the previously selected interface and setup the new one with same or different index/alternate settings. This function will allocate and initialize memory and data structures that are required to manage the specified interface. This includes creating a pipe bundle after opening the pipes for that interface. If the class for this interface is supported by the host stack then it will initialize that class. This function will also issue the device framework command (_usb_host_ch9_set_interface) to set the new interface on the device. When the application is notified of the completion of this command then the application/device-driver can issue class-specific commands or directly transfer data on the pipe.

Chapter 4
Device Framework Functions

4.1 USB Device Framework

This section describes the set of functions that are used to support device requests that are common for all USB devices.

For more information about USB Device framework, please refer to Chapter 9 of the USB 2.0 specification.

Table 4-1 summarizes the USB Device framework functions.

Table 4-1. Summary of USB Device framework functions

_usb_host_ch9_clear_feature _usb_host_ch9_get_configuration	Clear a specific feature Get device's current configuration value
_usb_host_ch9_get_descriptor	Get specified descriptor
_usb_host_ch9_get_interface	Get currently selected alternate setting for interface
_usb_host_ch9_get_status	Get status of specified recipient
_usb_host_ch9_set_address	Set device address
_usb_host_ch9_set_configuration	Set device configuration
_usb_host_ch9_set_descriptor	Set or update descriptors
_usb_host_ch9_set_feature	Set specific feature
_usb_host_ch9_set_interface	Set alternate interface settings
_usb_host_ch9_synch_frame	Set an endpoint's synchronization frame
_usb_hostdev_cntrl_request	Issue a class or vendor specific control request
_usb_host_register_ch9_callback	Register a callback function for a chapter 9 command

4.1.1 _usb_host_ch9_clear_feature

Clear a specific feature.

Synopsis

```
USB_STATUS  _usb_host_ch9_clear_feature(
  _usb_device_instance_handle dev_handle,
  uint_8                      req_type,
  uint_8                      intf_endpt,
  uint_16                  feature)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: Device, Interface or Endpoint)

intf_endpt [in] — The interface or endpoint number for this command

feature [in] — Feature selector such as Device remote wakeup, endpoint halt or test mode

Returns

USB_OK (success)

USBERR_INVALID_BMREQ_TYPE (failure; *req_type* is not valid)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_set_feature

Description — The function is used to clear or disable a specific feature on the specified device. Feature selector values must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

4.1.2 _usb_host_ch9_get_configuration

Get current configuration value for this device.

Synopsis

```
USB_STATUS _usb_host_ch9_get_configuration(
  _usb_device_instance_handle  dev_handle,
  uchar_ptr                    buffer)
```

Parameters

dev_handle [in] — USB device handle

buffer [out] — Configuration value

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_set_configuration

Description — The function returns the device's current configuration value. If the returned configuration value is zero then that means that the device is not configured.

4.1.3 _usb_host_ch9_get_descriptor

Get descriptor from this device.

Synopsis

```
USB_STATUS _usb_host_ch9_get_descriptor(
  _usb_device_instance_handle dev_handle,
  uint_16                     type_index,
  uint_16                     lang_id,
  uint_16                     buflen,
  uchar_ptr                   buffer)
```

Parameters

dev_handle [in] — USB device handle

type_index [in] — Type of descriptor and index

lang_id [in] — The language ID

buflen [in] — Buffer length

buffer [out] — Descriptor buffer

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_set_descriptor

Description — The device will return the specified descriptor if it exists. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device.

4.1.4 _usb_host_ch9_get_interface

Return the currently selected alternate setting for the specified interface.

Synopsis

```
USB_STATUS _usb_host_ch9_get_interface(
  _usb_device_instance_handle dev_handle,
  uint_8                      interface,
  uchar_ptr                   buffer)
```

Parameters

dev_handle [in] — USB device handle

interface [in] — Interface index

buffer [out] — Alternate setting buffer

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_set_interface

Description — The function allows the host to determine the currently selected alternate setting on the specified device.

4.1.5 _usb_host_ch9_get_status

Return status of the specified recipient.

Synopsis

```
USB_STATUS _usb_host_ch9_get_status(
  _usb_device_instance_handle dev_handle,
  uint_8                      req_type,
  uint_8                      intf_endpt,
  uchar_ptr                   buffer)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: Device, Interface or Endpoint)

intf_endpt [in] — The interface or endpoint number for this command

buffer [out] — Returned status

Returns

USB_OK (success)

USBERR_INVALID_BMREQ_TYPE (failure; *req_type* is not valid)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_clear_feature, _usb_host_ch9_set_feature, _usb_host_ch9_set_status

Description — The function returns the current status of the specified recipient.

4.1.6 _usb_host_ch9_set_address

Set the device address for device accesses.

Synopsis

```
USB_STATUS _usb_host_ch9_set_address(
  _usb_device_instance_handle dev_handle)
```

Parameters

dev_handle [in] — USB device handle

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

Description — The function sets the device address for all future device accesses.

4.1.7 _usb_host_ch9_set_configuration

Set device configuration.

Synopsis

```
USB_STATUS _usb_host_ch9_set_configuration(
  _usb_device_instance_handle dev_handle,
  uint_16                     config)
```

Parameters

dev_handle [in] — USB device handle

config [in] — Configuration value

Returns

USB_OK (success)
USBERR_DEVICE_NOT_FOUND (failure; device not found)
USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_set_configuration

Description — The function sets the device configuration. The lower byte of the configuration value specifies the desired configuration. This configuration value must be zero or match a configuration value from a configuration descriptor. If the configuration value is zero, the device is placed in its Address state. The upper byte of the configuration value is reserved.

4.1.8 _usb_host_ch9_set_descriptor

Update existing descriptor, or add new descriptors.

Synopsis

```
USB_STATUS _usb_host_ch9_set_descriptor(
  _usb_device_instance_handle dev_handle,
  uint_16                     type_index,
  uint_16                     lang_id,
  uint_16                     buflen,
  uchar_ptr                   buffer)
```

Parameters

dev_handle [in] — USB device handle

type_index [in] — Type of descriptor and index

lang_id [in] — The language ID

buflen [in] — Buffer length

buffer [out] — Descriptor buffer

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_get_descriptor

Description — This optional function issues a command that updates existing descriptors or adds new descriptors.

The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device.

4.1.9 _usb_host_ch9_set_feature

Set specified feature.

Synopsis

```
USB_STATUS _usb_host_ch9_set_feature(
  _usb_device_instance_handle dev_handle,
  uint_8                      req_type,
  uint_8                      intf_endpt,
  uint_16                  feature)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: Device, Interface or Endpoint)

intf_endpt [in] — The interface or endpoint number for this command

feature [in] — Feature selector such as Device remote wakeup, endpoint halt or test mode

Returns

USB_OK (success)

USBERR_INVALID_BMREQ_TYPE (failure; *req_type* is not valid)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_clear_feature

Description — This function will issue a command to set or enable a specified feature. Feature selector values must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

4.1.10 _usb_host_ch9_set_interface

Select an alternate setting for interface.

Synopsis

```
USB_STATUS _usb_host_ch9_set_interface(
  _usb_device_instance_handle dev_handle,
  uint_8                      alternate,
  uint_8                      intf)
```

Parameters

dev_handle [in] — USB device handle

alternate [in] — Alternate setting

intf [in] — Interface

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

_usb_host_ch9_get_interface

Description — This function allows the host to select an alternate setting for the specified interface.

4.1.11 _usb_host_ch9_synch_frame

Set and report an endpoint’s synchronization frame.

Synopsis

```
USB_STATUS _usb_host_ch9_synch_frame(
   _usb_device_instance_handle dev_handle,
   uint_8                      intf,
  uchar_ptr                    buffer)
```

Parameters

dev_handle [in] — USB device handle

intf [in] — Interface

buffer [out] — Synch frame buffer

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

Traits

See also

Description — This function is used to set and then report the endpoint’s synchronization frame. This command is relevant for isochronous endpoints only.

4.1.12 _usb_hostdev_cntrl_request

Issue a class or vendor specific control request.

Synopsis

```
USB_STATUS _usb_hostdev_cntrl_request(
  _usb_device_instance_handle  dev_handle,
  USB_SETUP_PTR                devreq,
  uchar_ptr                    buff_ptr,
  tr_callback                  callback,
  pointer                      callback param)
```

Parameters

dev_handle [in] — USB device

devreq [in] — Device request to send

buff_ptr [in] — Buffer to send/receive

callback [in] — Callback upon completion

callback param [in] — The parameter to pass back to the callback function

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

Description — This function is used to issue class- or vendor-specific control commands.

4.1.13 _usb_host_register_ch9_callback

Register a callback function for notification of standard device framework (chapter 9) command completion.

Synopsis

```
USB_STATUS _usb_host_register_ch9_callback(
  _usb_device_instance_handle  dev_handle,
  tr_callback                  callback,
  pointer                      callback param)
```

Parameters

dev_handle [in] — USB device

callback [in] — Callback upon completion

callback param [in] — The parameter to pass back to the callback function

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

Traits

See also

Description — This function registers a callback function that will be called to notify the user of a standard device framework request completion. This should be used only after enumeration is completed.

Chapter 5
Data Types

5.1 Data Type Descriptions

Table 5-1 describes the data types for compiler portability.

Table 5-1. Data types for Compiler Portability

Name	Bytes	Range		Description
		From	To	
boolean	4	0	NOT 0	0 = FALSE Non-zero = TRUE
pointer	4	0	0xFFFFFFFF	Generic pointer
PTR	4	0	0xFFFFFFFF	Generic pointer (*)
char	1	-127	127	Signed character
char_ptr	4	0	0xFFFFFFFF	Pointer to **char**
uchar	1	0	255	Unsigned character
uchar_ptr	4	0	0xFFFFFFFF	Pointer to **uchar**
int_8	1	-128	127	Signed character
int_8_ptr	4	0	0xFFFFFFFF	Pointer to **int_8**
uint_8	1	0	255	Unsigned character
uint_8_ptr	4	0	0xFFFFFFFF	Pointer to **uint_8**
int_16	2	-2^15	(2^15)-1	Signed 16-bit integer
int_16_ptr	4	0	0xFFFFFFFF	Pointer to **int_16**
uint_16	2	0	(2^16)-1	Unsigned 16-bit integer
uint_16_ptr	4	0	0xFFFFFFFF	Pointer to **uint_16**
int_32	4	-2^31	(2^31)-1	Signed 32-bit integer
int_32_ptr	4	0	0xFFFFFFFF	Pointer to **int_32**
uint_32	4	0	(2^32)-1	Unsigned 32-bit integer

Table 5-1. Data types for Compiler Portability (continued)

uint_32_ptr	4	0	0xFFFFFFFF	Pointer to **uint_32**
int_64	8	-2^63	(2^63)-1	Signed 64-bit integer
int_64_ptr	4	0	0xFFFFFFFF	Pointer to **int_64**
uint_64	8	0	(2^64)-1	Unsigned 64-bit integer
uint_64_ptr	4	0	0xFFFFFFFF	Pointer to **uint_64**
ieee_double	8	2.225074 E-308	1.7976923 E+308	Double-precision IEEE floating-point number
ieee_single	4	8.43E-37	3.37E+38	Single-precision IEEE floating-point number

Table 5-2 lists the USB Host API data types.

Table 5-2. USB Host API data types

USB Host API data Type	Simple Data Type
_usb_host_handle	pointer
_pipe_handle	pointer
_usb_device_instance_handle	pointer
_usb_interface_descriptor_handle	pointer

5.2 Data type Structures

5.2.1 PIPE_INIT_PARAM_STRUCT

Structure that defines the initialization parameters for a pipe; used by **_usb_host_open_pipe()**.

```
typedef struct
{
   pointer     DEV_INSTANCE;
   uint_32     INTERVAL;
   uint_32     MAX_PACKET_SIZE;
   uint_32     NAK_COUNT;
   uint_32     FIRST_FRAME;
   uint_32     FIRST_UFRAME;
   uint_32     FLAGS;
   uint_8      DEVICE_ADDRESS;
   uint_8      ENDPOINT_NUMBER;
   uint_8      DIRECTION;
   uint_8      PIPETYPE;
   uint_8      SPEED;
   uint_8      TRS_PER_UFRAME;
} PIPE_INIT_PARAM_STRUCT,  _PTR_ PIPE_INIT_PARAM_STRUCT_PTR;
```

Fields

DEV_INSTANCE —Instance of the device that owns this pipe.

INTERVAL — Interval for scheduling the data transfer on the pipe. For USB1.1, the value is in milliseconds. For USB 2.0, it is in 125-microsecond units.

MAX_PACKET_SIZE — Maximum packet size (in bytes) that the pipe is capable of sending or receiving.

NAK_COUNT — Maximum number of NAK responses per frame that are tolerated for the pipe. It is ignored for interrupt and isochronous pipes.

USB 1.1 — After NAK_COUNT NAK responses per frame, the transaction is deferred to the next frame.

USB 2.0 — The host controller does not execute a transaction if NAK_COUNT NAK responses are received on the pipe.

FIRST_FRAME — Frame number at which to start the transfer. If FIRST_FRAME equals 0, Host API schedules the transfer at the appropriate frame.

FIRST_UFRAME — Microframe number at which to start the transfer. If FIRST_FRAME equals 0, Host API schedules the transfer at the appropriate microframe.

FLAGS — One of:

- 0—(default) if the last data packet transferred is `MAX_PACKET_SIZE` bytes, terminate the transfer with a zero-length packet.
- 1—if the last data packet transferred is `MAX_PACKET_SIZE` bytes, do not terminate the transfer with a zero-length packet.

DEVICE_ADDRESS — Address of the USB device

DEVICE_ENDPOINT — Endpoint number of the device.

DIRECTION — Direction of transfer; one of:

- USB_RECV
- USB_SEND

PIPE_TYPE — Type of transfer to make on the pipe; one of:

- USB_BULK_PIPE
- USB_CONTROL_PIPE
- USB_INTERRUPT_PIPE
- USB_ISOCHRONOUS_PIPE

SPEED — Speed of transfer; one of:

- 0—full-speed transfer
- 1—low-speed transfer
- 2—high-speed transfer

TRS_PER_UFRAME — Number of transactions per microframe; one of:

- 1 (default)
- 2
- 3

If the field is 0, 1 is assumed. Applies to high-speed, high-bandwidth (USB 2.0) pipes only.

5.2.2 TR_INIT_PARAM_STRUCT

Transfer request; used as parameters to _usb_host_recv_data(), _usb_host_send_data(), and _usb_host_send_setup().

```
typedef struct
{
   uint_32     TR_INDEX;
   uchar_ptr   TX_BUFFER;
   uchar_ptr   RX_BUFFER;
   uint_32     TX_LENGTH;
   uint_32     RX_LENGTH;
   tr_callback CALLBACK;
   pointer     CALLBACK_PARAM;
   uchar_ptr   DEV_REQ_PTR;
} TR_INIT_PARAM_STRUCT,  TR_INIT_PARAM_STRUCT_PTR;
```

Fields

TR_INDEX — Transfer number on the pipe.

CONTROL_TX_BUFFER — Address of the buffer containing the data to be transmitted.

RX_BUFFER — Address of the buffer into which to receive data during the data phase.

TX_LENGTH — Length (in bytes) of data to be transmitted. For control transfers, it is the length of data for the data phase.

RX_LENGTH — Length (in bytes) of data to be received. For control transfers, it is the length of data for the data phase.

CALLBACK — The callback function to be invoked when a transfer is completed or an error is to be reported

CALLBACK_PARAM — The parameter to be passed back when the callback function is invoked.

DEV_REQ_PTR — Address of the setup packet to send. Applied to control pipes only.

5.2.3 USB_HOST_DRIVER_INFO

Information for one class or device driver, used by **_usb_host_driver_info_register**.

```
typedef struct driver_info
{
   uint_8          IDVENDOR[2];
   uint_8          IDPRODUCT[2];
   uint_8          BDEVICECLASS;
   uint_8          BDEVICESUBCLASS;
   uint_8          BDEVICEPROTOCOL;
   uint_8          RESERVED;
   event_callback ATTACH_CALL;
} USB_HOST_DRIVER_INFO, _PTR_ USB_HOST_DRIVER_INFO_PTR;
```

Fields

IDVENDOR[2] — Vendor ID per USB-IF

IDPRODUCT[2] — Product ID per manufacturer

BDEVICECLASS — Class code, if 0 see interface

BDEVICESUBCLASS — Sub-Class code, 0 if class = 0

BDEVICEPROTOCOL — Protocol, if 0 see interface

RESERVED — Alignment padding

ATTACH_CALL — The function to call when above information matches the one in device's descriptors occurs