Freescale MQX™ RTOS USB Host API Reference Manual

MQXUSBHOSTAPIRM
Rev. 5
02/2014

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXUSBHOSTAPIRM
Rev. 5, 02/2014

Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to freescale.com and navigate to Design Resources>Software and Tools>AllSoftware and Tools>Freescale MQX Software Solutions.

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 1	01/2009	Initial Release coming with MQX RTOS version 3.0
Rev. 2	12/2011	“USB Host Class API” and “Data Structures” chapters added.
Rev. 3	03/2013	“usb_class_mass_getmaxlun_bulkonly” section updated.
Rev. 4	04/2013	Minor edits.
Rev. 5	10/2013	Updated content to reflect the switch from MQX types to C99 types.

Chapter 1 Before You Begin

Chapter 2 Freescale MQX RTOS USB Host API Overview

Chapter 3 USB Host Layer API

Chapter 4 USB Chapter 9 Host API

Chapter 5 USB Host Class API

Chapter 6 Data Structures

Chapter 7 Data Types

Chapter 1 Before You Begin

1.1 About This Book

This document 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:

1.2 About MQX™ RTOS

MQX RTOS is real-time operating system from MQX Embedded. It has been designed for uniprocessor, multiprocessor, and distributed-processor embedded real-time systems.

To leverage the success of the MQX RTOS, Freescale Semiconductor adopted this software platform for its microprocessors. Comparing to the original MQX distributions, the Freescale MQX distribution was made simpler to configure and use. One single release now contains the MQX operating system plus all the other software components supported for a given microprocessor part such as network or USB communication stacks. The first MQX version released as Freescale MQX RTOS is assigned a number 3.0. It is based on and is API-level compatible with the MQX RTOS released by ARC at version 2.50.

In this book, we use MQX RTOS as the short name for MQX Real Time Operating System.

1.3 Acronyms and abbreviations

Table 1-1. Acronyms and abbreviations

Term	Description
API	Application Programming Interface
CDC	Communication Device Class
DCI	Device Controller Interface
HCI	Host Controller Interface

Table 1-1. Acronyms and abbreviations (continued)

HID	Human Interface Device
MSD	Mass Storage Device
MSC	Mass Storage Class
PHD	Personal Healthcare Device
PHDC	Personal Healthcare Device Class
QOS	Quality Of Service
SCSI	Small Computer System Interface
USB	Universal Serial Bus

1.4 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

1.5 Document Conventions

- Notes — Point out important information.

NOTE

Names of command-line options are case-sensitive.

- 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.

1.6 Function Listing Format

This is the general format of an entry for a function, compiler intrinsic, or macro.

function_name()

A short description of what function **function_name()** does.

Synopsis

Provides a prototype for function **function_name()**.

```
<return_type> function_name(
  <type_1>  parameter_1,
  <type_2>  parameter_2,
```

```
...
<type_n>  parameter_n)
```

Parameters

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

Parameter passing is categorized as follows:

- *In* — Means the function uses one or more values in the parameter you give it without storing any changes.
- *Out* — Means the function saves one or more values in the parameter you give it. You can examine the saved values to find out useful information about your application.
- *In/out* — Means the function changes one or more values in the parameter you give it and saves the result. You can examine the saved values to find out useful information about your application.

Returns

- Returns (success)
- Returns (failure)

Traits

Any of the following that might apply for the function:

- Blocks or the conditions under which it might block
- Must be started as a task
- 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 7, “Data Types.

Description

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

Chapter 2 Freescale MQX RTOS USB Host API Overview

2.1 Freescale MQX USB Host at a Glance

The USB Host stack provides USB 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 Host class library
- USB Chapter 9 requestor
- USB Host API—a hardware-independent application interface
- USB Host controller interface (HCI)—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 by 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 inter-operate 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.

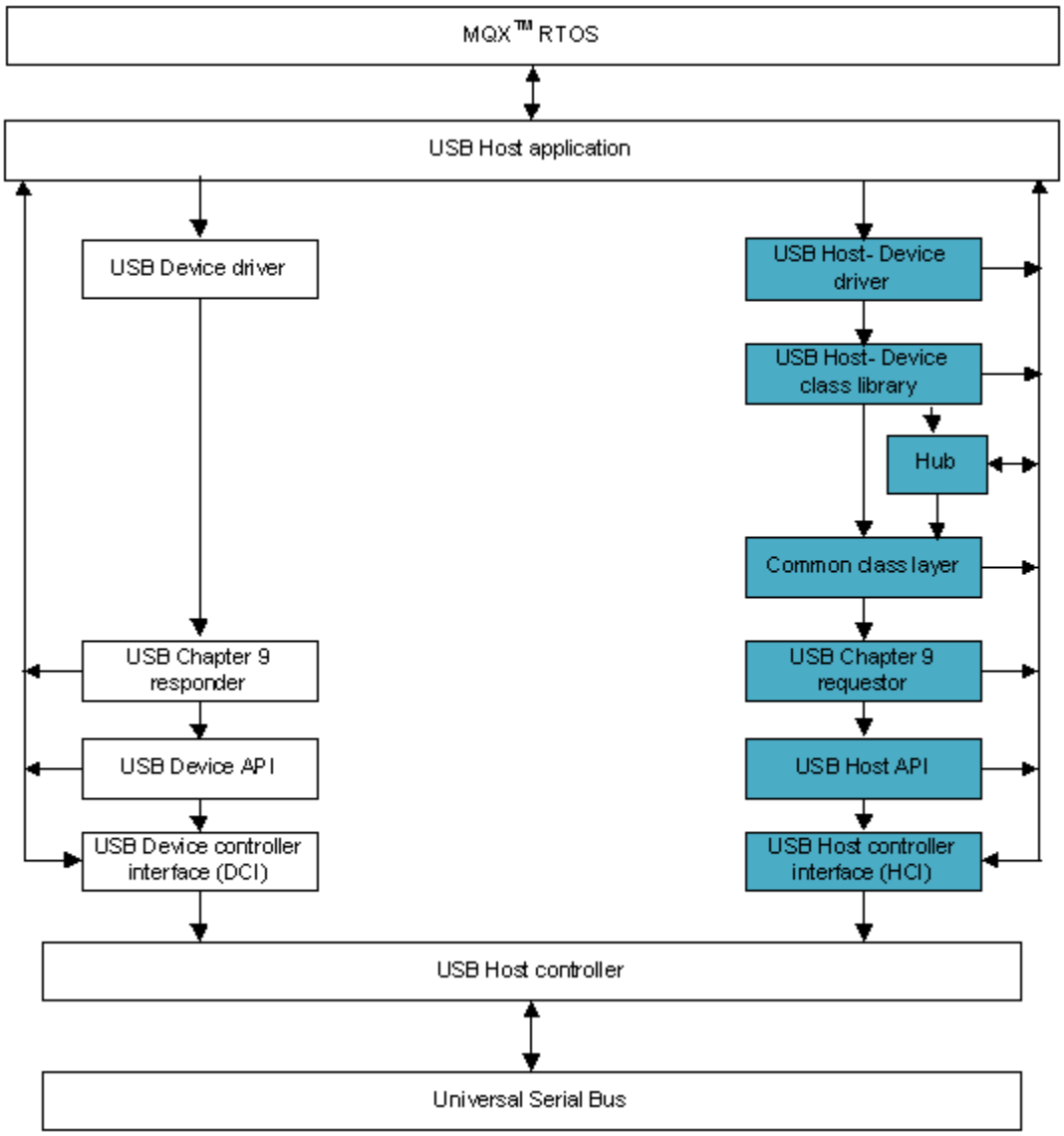

Figure 2-1. USB Host and USB Device Interaction

2.3 Using the Freescale MQX USB Host API

To use the Freescale MQX USB Host API, follow these general steps. Each API function is described in the subsequent 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 a 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, which is specified in the driver info 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 USB host stack locking mechanism

The USB host stack is ready for hot-plugging devices. When a device detach occurs, the event is propagated in the USB stack to the class driver which, in turn, deallocates memory space for the interface instance.

This can affect the application accessing the USB data and USB stack structures. In order to synchronize the interface deinitialization, the locking mechanism is used internally in the USB stack. The pair of *USB_lock()* and *USB_unlock()* functions defines the critical section in which the data are accessed and thus the detach event action is postponed. The locking mechanism uses internal counter so the pair of *USB_lock()* and *USB_unlock()* can be embedded:

```
USB_lock();
/* Entered critical section */
/* first check if the device is still valid or it was detached */
error = usb_hostdev_validate(dev_handle);
if (error == USB_OK) {
    /* Here we can ensure that the device was not detached so it is safe
    ** to access internal structures in the memory.
    */
    do_something();
}
/* About to leave critical section */
USB_unlock();
```

In the user application, there is no need to use the USB locking mechanism, if not explicitly requested. The application can handle the USB_DETACH_EVENT notification (see Section 2.3, “Using the Freescale MQX USB Host API). Also, it is strongly recommended to check the return values of API functions.

2.5 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 enqueueing the transfers. The enqueued transfer is then managed by hardware.

If using USB 2.0 hardware, the HCI 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 - provide the reliable, limited-latency delivery of data. The controller driver determines and allocates the required bandwidth. 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 low speed and full speed endpoints, the

interval is in milliseconds. For high speed endpoints, 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

2.6 USB Host API Summary

This section describes the list of USB host class API functions and their use. The following table summarizes the host layer API functions.

Table 2-1. Summary of Host Layer API functions

No.	API functions	Descriptor
1	**_usb_host_bus_control()**	Control the operation of the bus
2	**_usb_host_cancel_transfer()**	Cancel a specific transfer on a pipe
3	**_usb_host_close_all_pipes()**	Close all pipes
4	**_usb_host_close_pipe()**	Close a pipe
5	**_usb_host_driver_info_register()**	Register driver information
6	**_usb_host_get_frame_number()**	Get the current frame number
7	**_usb_host_get_micro_frame_number()**	Get the current microframe number
8	**_usb_host_get_transfer_status()**	Get the status of a specific transfer on a pipe
9	**_usb_host_init()**	Initialize the USB Host controller interface
10	**_usb_host_open_pipe()**	Open the pipe between a host and a device endpoint
11	**_usb_host_recv_data()**	Receive data on a pipe
12	**_usb_host_register_service()**	Register a service for a pipe or specific event
13	**_usb_host_send_data()**	Send data on a pipe
14	**_usb_host_send_setup()**	Send a setup packet on a control pipe
15	**_usb_host_shutdown()**	Shut down the USB Host controller interface
16	**_usb_host_unregister_service()**	Unregister a service for a pipe or specific event
17	**_usb_hostdev_find_pipe_handle()**	Find a pipe for the specified interface
18	**_usb_hostdev_get_buffer()**	Get a buffer for a particular device operation
19	**_usb_hostdev_get_buffer_aligned()**	Get an aligned budder for particular device operation
20	**_usb_hostdev_free_buffer()**	Free buffer associated with device

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

No.	API functions	Descriptor
21	**_usb_hostdev_get_descriptor()**	Get the specified USB descriptor that exists in device specific data structure
22	**_usb_hostdev_select_config()**	Select a new configuration of the device
23	**_usb_hostdev_select_interface()**	Select a new interface on the device

The following table summarizes the USB device framework functions.

Table 2-2. Summary of Device framework functions

No.	API functions	Descriptor
1	**_usb_host_ch9_clear_feature()**	Clears a specific feature
2	**_usb_host_ch9_get_configuration()**	Gets device's current configuration value
3	**_usb_host_ch9_get_descriptor()**	Gets specified descriptor
4	**_usb_host_ch9_get_interface()**	Gets currently selected alternate setting for interface
5	**_usb_host_ch9_get_status()**	Gets status of the specified recipient
6	**_usb_host_ch9_set_address()**	Sets device address
7	**_usb_host_ch9_set_configuration()**	Sets device configuration
8	**_usb_host_ch9_set_descriptor()**	Sets or updates descriptors
9	**_usb_host_ch9_set_feature()**	Sets specific feature
10	**_usb_host_ch9_set_interface()**	Sets alternate interface settings
11	**_usb_host_ch9_synch_frame()**	Sets an endpoint's synchronization frame
12	**_usb_hostdev_cntrl_request()**	Issues a class or vendor specific control request
13	**_usb_host_register_ch9_callback()**	Registers a callback function for a chapter 9 command

The following table summarizes the AUDIO class API functions.

Table 2-3. Summary of Audio class API functions

No.	API functions	Descriptor
1	**usb_class_audio_control_init()**	Initializes the class driver for audio control interface.
2	**usb_class_audio_stream_init()**	Initializes the class driver for audio stream interface.
3	**usb_class_audio_control_get_descriptors()**	The function searches for descriptors of audio control interface.
4	**usb_class_audio_control_set_descriptors()**	Set descriptors for audio control interface.
5	**usb_class_audio_stream_get_descriptors()**	This function searches for descriptors of audio stream interface.
6	**usb_class_audio_stream_set_descriptors()**	Set descriptors for audio stream interface.
7	**usb_class_audio_init_ipipe()**	Starts interrupt endpoint to poll for interrupt on specified device.
8	**usb_class_audio_recv_data()**	Receives audio data from the isochronous IN pipe

Table 2-3. Summary of Audio class API functions (continued)

No.	API functions	Descriptor
9	usb_class_audio_send_data()	Sends audio data to the isochronous OUT pipe
10	usb_class_audio_[specific_request]()	Sends specific requests to device through control end point

The following table summarizes the CDC class API functions.

Table 2-4. Summary of CDC class API functions

No.	API functions	Descriptor
1	usb_class_cdc_acm_init()	Initializes the class driver for Abstract Class Control.
2	usb_class_cdc_bind_acm_interface()	Binds data interface to appropriate control interface.
3	usb_class_cdc_bind_data_interfaces()	Binds all data interfaces belonging to ACM control instance.
4	usb_class_cdc_data_init()	Initializes the class driver for Abstract Class Data.
5	usb_class_cdc_get_acm_descriptors()	Search for descriptors in the device configuration and fills back fields if the descriptor was found.
6	usb_class_cdc_get_acm_line_coding()	Gets parameters of current line.
7	usb_class_cdc_get_ctrl_descriptor()	Search for descriptor of control interface, which controls data interface identified by descriptor (intf_handle).
8	usb_class_cdc_get_ctrl_interface()	Finds registered control interface in the chain.
9	usb_class_cdc_get_data_interface()	Finds registered data interface in the chain.
10	usb_class_cdc_init_ipipe()	Starts interrupt endpoint to poll for interrupt on specified device.
11	usb_class_cdc_install_driver()	Adds/installs USB serial device driver.
12	usb_class_cdc_set_acm_ctrl_state()	This function is used to set parameters of current line (baud rate, HW control, and so on).
13	usb_class_cdc_set_acm_descriptors()	Sets descriptors for ACM interface.
14	usb_class_cdc_set_acm_line_coding()	Sets parameters of current line.
15	usb_class_cdc_unbind_acm_interface()	Unbinds data interface to appropriate control interface.
16	usb_class_cdc_unbind_data_interfaces()	Unbinds all data interfaces bound to ACM control instance.
17	usb_class_cdc_uninstall_driver()	Removes USB serial device driver.

The following table summarizes the HID class API functions.

Table 2-5. Summary of HID class functions

No.	API functions	Descriptor
1	usb_class_hid_get_idle()	Reads the idle rate of a particular HID device report.
2	usb_class_hid_get_protocol()	Reads the active protocol.
3	usb_class_hid_get_report()	Gets a report from the HID device.

Table 2-5. Summary of HID class functions (continued)

No.	API functions	Descriptor
4	usb_class_hid_init()	Initializes the class driver.
5	usb_class_hid_set_idle()	Silences a particular report on interrupt in pipe until a new event occurs or specified time elapses.
6	usb_class_hid_set_protocol()	Switches between the boot protocol and the report protocol (or vice versa).
7	usb_class_hid_set_report()	Sends a report to the HID device.

The following table summarizes the MSD class API functions.

Table 2-6. Summary of MSD class API functions

No.	API functions	Descriptor
1	usb_class_mass_getmaxlun_bulkonly()	Gets the number of logical units on the device.
2	usb_class_mass_init()	Initializes the mass storage class.
3	usb_class_mass_reset_recovery_on_usb()	Gets the pending request from class driver queue and sends the RESET command on control pipe.
4	usb_class_mass_storage_device_command()	Executes the command defined in protocol API.
5	usb_class_mass_storage_device_command_cancel()	Dequeues the command in class driver queue.
6	usb_class_mass_cancelq()	Cancels the given request in the queue.
7	usb_class_mass_deleteq()	Deletes the pending request in the queue.
8	usb_class_mass_get_pending_request()	Fetches the pointer to the first (pending) request in the queue, or NULL if there is no pending requests.
9	usb_class_mass_q_init()	Initializes a mass storage class queue.
10	usb_class_mass_q_insert()	Inserts a command in the queue.
11	usb_mass_ufi_cancel()	This function cancels the given request in the queue.
12	usb_mass_ufi_generic()	This function initializes the mass storage class.

The following table summarizes the PHDC class API functions.

Table 2-7. Summary of PHDC class API functions

No.	API functions	Descriptor
1	usb_class_phdc_init()	This function serves the main purpose of initializing the PHDC interface structure
2	usb_class_phdc_set_callbacks()	The function is used to register the application defined callback functions for the PHDC send, receive, and control request actions
3	usb_class_phdc_send_control_request()	The function is used to send PHDC class specific request to the attached device

Table 2-7. Summary of PHDC class API functions

No.	API functions	Descriptor
4	**usb_class_phdc_recv_data()**	The function is used for receiving PHDC class specific data or metadata packets
5	**usb_class_phdc_send_data()**	The function is used for sending PHDC class specific data or metadata packets
6	**usb_class_phdc_cancel_transfer()**	The function attempts to cancel the indicated transfer

Chapter 3 USB Host Layer API

3.1 USB Host Layer API function listing

3.1.1 _usb_host_bus_control()

Control the operation of the bus.

Synopsis

```
void  _usb_host_bus_control
(
        usb_host_handle      hci_handle,
        uint8_t               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 de-asserting the bus reset, asserting and de-asserting resume signalling, suspending and resuming the SOF generation.

3.1.2 _usb_host_cancel_transfer()

Cancel the specified transfer on the pipe.

Synopsis

```
uint32_t  _usb_host_cancel_transfer
(
    _usb_host_handle  host_handle,
    _usb_pipe_handle  pipe_handle,
    uint32_t           transfer_number
)
```

Parameters

host_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 (**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 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.

3.1.3 _usb_host_close_all_pipes()

Close all pipes.

Synopsis

```
void  _usb_host_close_all_pipes
(
       _usb_host_handle_       host_handle
)
```

Parameters

host_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.

3.1.4 _usb_host_close_pipe()

Close the specified pipe functions.

Synopsis

```
uint32_t  _usb_host_close_pipe
(
    _usb_host_handle_      host_handle,
    _usb_pipe_handle       pipe_handle
)
```

Parameters

host_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 all pipes from the list of open pipes.

3.1.5 _usb_host_driver_info_register()

Register driver information.

Synopsis

```
USB_STATUS _usb_host_driver_info_register
(
    _usb_host_handle      host_handle,
    void *                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.

3.1.6 _usb_host_get_frame_number()

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

Synopsis

```
uint32_t  _usb_host_get_frame_number
(
    _usb_host_handle       host_handle
)
```

Parameters

host_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.

3.1.7 _usb_host_get_micro_frame_number()

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

Synopsis

```
uint32_t  _usb_host_get_micro_frame_number
(
    _usb_host_handle      host_handle
)
```

Parameters

host_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.

3.1.8 _usb_host_get_transfer_status()

Get the status of the specified transfer on the pipe.

Synopsis

```
uint32_t  _usb_host_get_transfer_status
(
    usb_pipe_handle                 pipe_handle,
    uint32_t                        transfer_number
)
```

Parameters

pipe_handle [in] — Pipe handle

transfer_number [in] — Specific transfer number on the pipe

Should correspond to 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**.

3.1.9 _usb_host_init()

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

Synopsis

```
uint32_t  _usb_host_init
(
    uint8_t                     devnum,
    uint32_t                    frame_list_size,
    _usb_host_handle  *         host_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)

host_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 an HCI 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).

3.1.10 _usb_host_open_pipe()

Open a pipe between the host and the device endpoint.

Synopsis

```
uint32_t  _usb_host_open_pipe
(
    _usb_host_handle               host_handle,
    PIPE_INIT_PARAM_STRUCT_PTR     pipe_init_params_ptr,
    _usb_pipe_handle  *            pipe_handle
)
```

Parameters

host_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

3.1.11 _usb_host_recv_data()

Receive data on a pipe.

Synopsis

```
uint32_t  _usb_host_recv_data
(
    _usb_host_handle            host_handle,
    _usb_pipe_handle            pipe_handle,
    TR_INIT_PARAM_STRUCT_PTR    tr_params_ptr
)
```

Parameters

host_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 an HCI 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.

3.1.12 _usb_host_register_service()

Register a service for a specific event.

Synopsis

```
uint32_t  _usb_host_register_service
(
    _usb_host_handle  host_handle,
    uint8_t            type,
    void (_CODE_PTR_ service)(void *  callbk_ptr,
                              uint32_t   event_param)
)
```

Parameters

host_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 an 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.

3.1.13 _usb_host_send_data()

Send data on a pipe.

Synopsis

```
uint32_t  _usb_host_send_data
(
    _usb_host_handle              host_handle,
    _usb_pipe_handle              pipe_handle,
    TR_INIT_PARAM_STRUCT_PTR      tr_params_ptr
)
```

Parameters

host_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 an HCI 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 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 which is a 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.

3.1.14 _usb_host_send_setup()

Send a setup packet on a control pipe functions.

Synopsis

```
uint32_t  _usb_host_send_setup
(
    _usb_host_handle            host_handle,
    _usb_pipe_handle            pipe_handle,
    TR_INIT_PARAM_STRUCT_PTR    tr_params_ptr
)
```

Parameters

host_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 an HCI function to queue the transfer and then returns. Once a control transfer request is queued, the HCI 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**.

3.1.15 _usb_host_shutdown()

Shut down the USB Host controller interface.

Synopsis

```
void  _usb_host_shutdown
(
    _usb_host_handle                host_handle
)
```

Parameters

host_handle [in] — USB Host controller handle

Returns

Traits

See also

_usb_host_init()

Description

The function calls an HCI 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 re-configured.

The function additionally does the following:

1. Terminates all transfers.
2. Un-registers all services.
3. Disconnects the host from the USB bus.
4. Frees all memory that the USB Host allocated for its internal data.

3.1.16 _usb_host_unregister_service()

Un-register a service for a type of event.

Synopsis

```
uint32_t _usb_host_unregister_service
(
    _usb_host_handle              host_handle,
    uint8_t                       event
)
```

Parameters

host_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 un-registers the callback function that services the event. As a result, the event can no longer be serviced by a callback function.

3.1.17 _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,
    _uint8_t                        pipe_type,
    _uint8_t                        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** (failure)

Traits

See also

_usb_hostdev_select_interface()

Description

This function is used to find an open 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()**, NULL is returned.

3.1.18 _usb_hostdev_get_buffer()

Get a buffer for the device operation.

Synopsis

```
USB_STATUS _usb_hostdev_get_buffer
(
    _usb_device_instance_handle     dev_handle,
    uint32_t                        buffer_size,
    unsigned char                   **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

_usb_hostdev_get_buffer_aligned(), _usb_hostdev_free_buffer()

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 freed by the host system software.

3.1.19 _usb_hostdev_get_buffer_aligned()

Get a buffer for the device operation.

Synopsis

```
USB_STATUS _usb_hostdev_get_buffer
(
    _usb_device_instance_handle     dev_handle,
    uint32_t                        buffer_size,
    uint32_t                         alignement,
    unsigned char                   **buff_ptr
)
```

Parameters

dev_handle [in] — USB device

buffer size [in] — Buffer size to get

alignment [in] — Alignment size

buff_ptr [out] — Pointer to the buffer

Returns

- Pointer to the buffer (success)
- **USBERR_DEVICE_NOT_FOUND** (failure; device not found)

Traits

See also

_usb_hostdev_get_buffer(), _usb_hostdev_free_buffer()

Description

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

3.1.20 _usb_hostdev_free_buffer()

Get a buffer for the device operation.

Synopsis

```
USB_STATUS _usb_hostdev_get_buffer
(
    _usb_device_instance_handle     dev_handle,
    unsigned char                   **buff_ptr
)
```

Parameters

dev_handle [in] — USB device

buff_ptr [out] — Pointer to the buffer

Returns

- **USBERR_DEVICE_NOT_FOUND** (failure; device not found)

Traits

See also

_usb_hostdev_get_buffer(), _usb_hostdev_get_buffer_aligned()

Description

Applications should use this function to free buffer allocated with MQX USB Host API functions that will no longer be used by an application.

3.1.21 _usb_hostdev_get_descriptor()

Get a descriptor.

Synopsis

```
USB_STATUS _usb_hostdev_get_descriptor
(
    _usb_device_instance_handle             dev_handle,
    usb_interface_descriptor_handle         intf_handle,
    descriptor_type                         desc_type,
    uint8_t                                 desc_index,
    uint8_t                                 intf_alt,
    void                                    **descriptor
)
```

Parameters

dev_handle [in] — USB device

intf_handle[in] — interface descriptor handle

desc_type [in] — The type of descriptor to get

desc_index [in] — The descriptor index

intf_alt [in] — The interface alternate

**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 the interface and endpoint descriptors, and stores them in the internal device-specific memory. The application can request the descriptors by calling this function instead of issuing a device framework function request to get the descriptor from the device.

The descriptor is searched by following these criterions:

1. If the *desc_type* is USB_DESC_TYPE_DEV, the returned descriptor is device descriptor.

2. If the *desc_type* is USB_DESC_TYPE_CFG, the returned descriptor is configuration descriptor.

3. If the *desc_type* is USB_DESC_TYPE_IF, the returned *desc_index* descriptor is interface descriptor.

4. If the *intf_handle* is NULL, the descriptor *desc_type* is searched within the whole configuration.

5. If the *intf_handle* is not NULL, the descriptor *desc_type* is searched within the interface.

3.1.22 _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_t                         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, it will delete that configuration and select a new one. The host system sends a device framework command (**_usb_host_ch9_get_configuration()**) to the device and then initializes and saves the configuration specific information in its internal data structures.

3.1.23 _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,
    void *                               class_intf_ptr
)
```

Parameters

dev_handle [in] — USB device

intf_handle [in] — Interface to be selected

class_intf_ptr [out] — Initialized class-specific interface structure

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 set up a new one with the 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, 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, the application/device-driver can issue class-specific commands or directly transfer data on the pipe.

Chapter 4 USB Chapter 9 Host API

4.1 USB Host Chapter 9 API function listing

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,
        uint8_t                     req_type,
        uint8_t                     intf_endpt,
        uint16_t                    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 wake-up, 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,
    unsigned char                   *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 current configuration value of the device. If the returned configuration value is zero, it 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,
                uint16_t                    type_index,
                uint16_t                    lang_id,
                uint16_t                    buflen,
                unsigned char               *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 requested descriptor value *type_index* contains 16 bit value USB_DESC_TYPE_xxx, the higher byte, and the index, the lower byte. 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,
    uint8_t                     interface,
    unsigned char               *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,
    uint8_t                     req_type,
    uint8_t                     intf_endpt,
    unsigned char               *buffer
)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: REQ_TYPE_DEVICE, REQ_TYPE_INTERFACE or REQ_TYPE_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()**

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,
    uint16_t                    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_get_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 the existing descriptor, or add new descriptors.

Synopsis

```
USB_STATUS _usb_host_ch9_set_descriptor
(
    _usb_device_instance_handle     dev_handle,
    uint16_t                        type_index,
    uint16_t                        lang_id,
    uint16_t                        buflen,
    unsigned char                   *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 requested descriptor value *type_index* contains a 16 bit value USB_DESC_TYPE_xxx, the higher byte, and the index, the lower byte. 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,
    uint8_t                     req_type,
    uint8_t                     intf_endpt,
    uint16_t                    feature
)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: REQ_TYPE_DEVICE, REQ_TYPE_INTERFACE or REQ_TYPE_ENDPOINT)

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

feature [in] — Feature selector such as defined in Chapter 9

of the USB 2.0 Specification 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 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,
    uint8_t                     alternate,
    uint8_t                     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 the synchronization frame of an endpoint.

Synopsis

```
USB_STATUS _usb_host_ch9_synch_frame
(
    _usb_device_instance_handle dev_handle,
    uint8_t                     intf,
    unsigned char               *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 synchronization frame of an endpoint. 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,
    unsigned char                *buff_ptr,
    tr_callback                  callback,
    void                          *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-specific 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,
    void                          *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 USB Host Class API

5.1 Audio Class API Function Listing

This section defines the API functions used for the Audio Device Class. The application can use these API functions to make Audio applications.

5.1.1 usb_class_audio_control_init()

Initialize the class driver for audio control interface.

Synopsis

```
void usb_class_audio_control_init
(
    PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
    CLASS_CALL_STRUCT_PTR ccs_ptr,
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Returns

Traits

See also

CLASS_CALL_STRUCT_PTR

PIPE_BUNDLE_STRUCT_PTR

Description

This function is called by a common class to initialize the class driver for the audio stream interface. It is called in response to a select interface which is called by an application.

5.1.2 usb_class_audio_stream_init()

Initialize the class driver for audio stream interface.

Synopsis

```
void usb_class_audio_stream_init
(
    PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
    CLASS_CALL_STRUCT_PTR ccs_ptr,
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR

PIPE_BUNDLE_STRUCT_PTR

Description

This function is called by a common class to initialize the class driver for the audio stream interface. It is called in response to a select interface which is called by an application.

5.1.3 usb_class_audio_control_get_descriptors()

The function is searching for descriptors of the audio control interface.

Synopsis

```
uint8_t usb_class_audio_control_get_descriptors
(
    _usb_device_instance_handle                         dev_handle,
    _usb_interface_descriptor_handle          intf_handle,
    USB_AUDIO_CTRL_DESC_HEADER_PTR  * header_desc,
    USB_AUDIO_CTRL_DESC_IT_PTR  * it_desc,
    USB_AUDIO_CTRL_DESC_OT_PTR  * ot_desc,
    USB_AUDIO_CTRL_DESC_FU_PTR  * fu_desc,
)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

header_desc [out] — Pointer to header functional descriptor

it_desc [out] — Pointer to input terminal descriptor

ot_desc [out] — Pointer to output terminal descriptor

fu_desc [out] — Pointer to feature unit descriptor

Returns

- **USB_OK** (success)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)

Traits

See also

usb_class_audio_control_set_descriptors()

USB_AUDIO_CTRL_DESC_HEADER_PTR

USB_AUDIO_CTRL_DESC_IT_PTR

USB_AUDIO_CTRL_DESC_OT_PTR

USB_AUDIO_CTRL_DESC_FU_PTR

Description

This function is searching for descriptors of audio control interface and fills back fields if the descriptor was found.

5.1.4 usb_class_audio_control_set_descriptors()

Set descriptors for audio control interface.

Synopsis

```
USB_STATUS usb_class_audio_control_set_descriptors
(
    CLASS_CALL_STRUCT_PTR ccs_ptr,
    USB_AUDIO_CTRL_DESC_HEADER_PTR                    header_desc,
    USB_AUDIO_CTRL_DESC_IT_PTR it_desc,
    USB_AUDIO_CTRL_DESC_OT_PTR ot_desc,
    USB_AUDIO_CTRL_DESC_FU_PTR fu_desc,
)
```

Parameters

ccs_ptr [out] — The communication device data instance structure

header_desc [in] — Pointer to header functional descriptor

it_desc [in] — Pointer to input terminal descriptor

ot_desc [in] — Pointer to output terminal descriptor

fu_desc [in] — Pointer to unit descriptor

Returns

USB_OK (if validation passed)

Traits

See also

usb_class_audio_control_get_descriptors()

CLASS_CALL_STRUCT_PTR

USB_AUDIO_CTRL_DESC_HEADER_PTR

USB_AUDIO_CTRL_DESC_IT_PTR

USB_AUDIO_CTRL_DESC_OT_PTR

USB_AUDIO_CTRL_DESC_FU_PTR

Description

Set descriptors for the audio control interface. Descriptors can be used afterwards either by an application or by the driver.

5.1.5 usb_class_audio_stream_get_descriptors()

This function is searching for descriptors of the audio stream interface.

Synopsis

```
uint8_t usb_class_audio_stream_get_descriptors
(
    _usb_device_instance_handle                          dev_handle,
    _usb_interface_descriptor_handle            intf_handle,
    USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR  * as_itf_desc,
    USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR  *    frm_type_desc,
    USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR  * iso_endp_spec_desc,
)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

as_itf_desc [out] — Pointer to specific audio stream interface descriptor

frm_type_desc [out] — Pointer to format type descriptor

iso_endp_spec_desc [out] — Pointer to specific isochronous endpoint descriptor

Returns

- **USB_OK** (success)
- **USBERR_INIT_FAILED** (failure: device initialization failed)

Traits

See also

usb_class_audio_stream_set_descriptors()

USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR

USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR

USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

Description

This function is searching for descriptors of the audio stream interface and fills back fields if the descriptor was found.

5.1.6 usb_class_audio_stream_set_descriptors()

Set descriptors for the audio stream interface.

Synopsis

```
USB_STATUS usb_class_audio_stream_set_descriptors
(
    CLASS_CALL_STRUCT_PTR                       ccs_ptr,
    USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR    as_itf_desc,
    USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR       frm_type_desc,
    USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR iso_endp_spec_desc,
)
```

Parameters

ccs_ptr [out] — The communication device data instance structure

as_itf_desc [in] — Pointer to audio stream specific interface descriptor

frm_type_desc [in] — Pointer to format type descriptor

iso_endp_spec_desc [in] — Pointer to isochronous endpoint specific descriptor

Returns

- **USB_OK** (if validation passed)
- Others (failure)

Traits

See also

usb_class_audio_stream_get_descriptors()

CLASS_CALL_STRUCT_PTR

USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR

USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR

USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

Description

This function sets descriptors for the audio stream interface. Descriptors can be used afterwards either by an application or by the driver.

5.1.7 usb_class_audio_init_ipipe()

Start interrupt endpoint to poll for interrupt on specified device.

Synopsis

```
USB_STATUS usb_class_audio_init_ipipe
(
    CLASS_CALL_STRUCT_PTR audio_instance,
    tr_callback user_callback,
    void *user_callback_param
)
```

Parameters

audio_instance [in] — Audio control interface instance

user_callback [in] — User callback function

user_callback_param [in] — User callback parameter

Returns

- **USB_OK** (success)
- **USBERR_OPEN_PIPE_FAILED** (failure: interrupt pipe is NOT found)

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

The function starts interrupt endpoint to poll for interrupt on a specified device.

5.1.8 usb_class_audio_recv_data()

Receive audio data from the isochronous IN pipe.

Synopsis

```
USB_STATUS usb_audio_recv_data
(
    CLASS_CALL_STRUCT_PTR control_ptr,
    CLASS_CALL_STRUCT_PTR stream_ptr,
    tr_callback callback,
    void *call_param,
    uint32_t buf_size,
    void *buffer
)
```

Parameters

control_ptr [in] — Class-interface control pointer

stream_ptr [in] — Class-interface stream pointer

callback [in] — Callback upon completion

call_param [in] — user parameter returned by callback

buf_size [in] — data length

buffer [out] — pointer to received buffer

Returns

- **USB_OK/USB_STATUS_TRANSFER_QUEUED** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_OPEN_PIPE_FAILED** (isochronous pipe is NULL)
- **USBERR_INVALID_PIPE_HANDLE** (pipe ID is invalid)

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

This function is used for receiving audio data from isochronous IN pipe. Before scheduling the receive action, this function will first validate the provided class-interface control pointer then checking isochronous IN pipe. If all checking is passed, the function initiates a USB host receive action on the designated pipe and registers a callback function to application.

5.1.9 usb_class_audio_send_data()

Send audio data to the isochronous OUT pipe.

Synopsis

```
USB_STATUS usb_audio_send_data
(
    CLASS_CALL_STRUCT_PTR control_ptr,
    CLASS_CALL_STRUCT_PTR stream_ptr,
    tr_callback callback,
    void *call_param,
    uint32_t buf_size,
    void *buffer
)
```

Parameters

control_ptr [in] — Class-interface control pointer

stream_ptr [in] — Class-interface stream pointer

callback [in] — Callback upon completion

call_param [in] — user parameter returned by callback

buf_size [in] — data length

buffer [in] — send buffer pointer

Returns

- **USB_OK/USB_STATUS_TRANSFER_QUEUED** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_OPEN_PIPE_FAILED** (isochronous pipe is NULL)
- **USBERR_INVALID_PIPE_HANDLE** (pipe ID is invalid)

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

This function is used for sending audio data from the isochronous OUT pipe. Before scheduling the send action, this function will first validate the provided class-interface control pointer, and it will then check the isochronous OUT pipe. If all checking is passed, the function initiates a USB host send action on the designated pipe and registers a callback function to an application.

5.1.10 usb_class_audio_[specific_request]()

USB host class driver provides the ability to send the following specific requests:

- Copy Protect Control
- Mute Control
- Volume Control (CUR, MIN, MAX, RES)
- Bass Control (CUR, MIN, MAX, RES)
- Mid Control (CUR, MIN, MAX, RES)
- Treble Control (CUR, MIN, MAX, RES)
- Graphic Eq Control (CUR, MIN, MAX, RES)
- Automatic Gain Control
- Delay Control (CUR, MIN, MAX, RES)
- Bass Boost Control
- Sampling Frequency Control (CUR, MIN, MAX, RES)
- Pitch Control
- Memory

Each request includes two Get/Set individual functions. General format of all of these functions (except: Get/Set Graphic Eq Control and Get/Set Memory) is described below.

Synopsis

```
USB_STATUS usb_class_audio_<request_name>
(
    AUDIO_COMMAND_PTR command_ptr,
    void *buf,
)
```

Parameters

command_ptr [in] — Class interface structure pointer

buf [in] — Buffer to receive data

Returns

- **USB_OK** if command has been passed on USB
- Others (failure)

Traits

See also

HID_COMMAND_PTR

Description

The function is used for sending a specific request to the attached device.

NOTE

usb_class_audio_get/set_graphic_eq and *usb_class_audio_get/set_mem_endpoint* functions have more input parameters than the general form. A blen, buffer length parameter, needs to be added in *usb_class_audio_get/set_graphic_eq functions*. The blen and offset, zero-offset parameters, need to be added in the *usb_class_audio_get/set_mem_endpoint* functions.

5.2 CDC Class API Function Listing

This section defines the API functions used for the Communication Device Class (CDC). The application can use these API functions to make CDC applications.

5.2.1 usb_class_cdc_acm_init()

Initialize the class driver for AbstractClassControl.

Synopsis

```
void usb_class_cdc_acm_init
(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface.

ccs_ptr [in] — The communication device data instance structure

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR, **PIPE_BUNDLE_STRUCT_PTR**

Description

This function is called by a common class to initialize the class driver for AbstractClassControl. It is called in response to a select interface call by an application.

5.2.2 usb_class_cdc_bind_acm_interface()

Bind data interface to the appropriate control interface.

Synopsis

```
USB_STATUS usb_class_cdc_bind_acm_interface
(
        CLASS_CALL_STRUCT_PTR ccs_ptr,
        INTERFACE_DESCRIPTOR_PTR if_desc
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

if_desc [in] — Interface descriptor pointer

Returns

USB_OK

Traits

See also

usb_class_cdc_unbind_acm_interface(), **CLASS_CALL_STRUCT_PTR**,

INTERFACE_DESCRIPTOR_PTR

Description

Data interface, which is specified by ccs_ptr, will be bound to the appropriate control interface. The function must be called in the USB host stack locked state and validated USB device.

5.2.3 usb_class_cdc_bind_data_interfaces()

Bind all data interfaces belonging to the ACM control instance.

Synopsis

```
USB_STATUS usb_class_cdc_bind_data_interfaces
(
        _usb_device_instance_handle dev_handle,
        CLASS_CALL_STRUCT_PTR ccs_ptr
)
```

Parameters

dev_handle [in] — Pointer to device instance

ccs_ptr [in] — The communication device data instance structure

Returns

USB_ OK (if found)

Traits

See also

usb_class_cdc_unbind_data_interfaces(), **CLASS_CALL_STRUCT_PTR**

Description

All data interfaces, which belong to the ACM control instance and are specified by the ccs_ptr, will be bound to this interface. The union functional descriptor describes which data interfaces should be bound. This function must be called in the USB host stack locked state and validated USB device.

5.2.4 usb_class_cdc_data_init()

Synopsis

```
void usb_class_cdc_data_init
(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR, **PIPE_BUNDLE_STRUCT_PTR**

Description

This function is called by a common class to initialize the class driver for AbstractClassControl. It is called in response to a select interface call by an application.

5.2.5 usb_class_cdc_get_acm_descriptors()

Search for descriptors in the device configuration and fill back fields if the descriptor was found.

Synopsis

```
USB_STATUS usb_class_cdc_get_acm_descriptors
(
        _usb_device_instance_handle dev_handle,
        _usb_interface_descriptor_handle intf_handle,
        USB_CDC_DESC_ACM_PTR  * acm_desc,
        USB_CDC_DESC_CM_PTR  * cm_desc,
        USB_CDC_DESC_HEADER_PTR  * header_desc,
        USB_CDC_DESC_UNION_PTR  * union_desc
)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

acm_desc [in] — ACM functional descriptor pointer

cm_desc [in] — CM functional descriptor pointer

header_desc [in] — Header functional descriptor pointer

union_desc [in] — Union functional descriptor pointer

Returns

- **USB_OK** (success)
- **Others** (failure)

Traits

See also

usb_class_cdc_set_acm_descriptors(),**USB_HOST_DRIVER_INFO**,**USB_CDC_DESC_CM_PTR**, **USB_CDC_DESC_HEADER_PTR**, **USB_CDC_DESC_UNION_PTR**

Description

This function is searching for descriptors in the device configuration and filling back fields if the descriptor was found. The function must be called in the USB host stack locked state and validated USB device.

5.2.6 usb_class_cdc_get_acm_line_coding()

Get parameters of the current line.

Synopsis

```
USB_STATUS usb_class_cdc_get_acm_line_coding
(
        CLASS_CALL_STRUCT_PTR     ccs_ptr,
        USB_CDC_UART_CODING_PTR   uart_coding_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

uart_coding_ptr [in] — Location to store coding into

Returns

Success as **USB_OK**

Traits

See also

usb_class_cdc_set_acm_line_coding()

CLASS_CALL_STRUCT_PTR

USB_CDC_UART_CODING_PTR

Description

This function is used to get parameters of the current line such as baud rate, HW control, etc.

NOTE

Although this function issues a command to the CDC control interface, the parameter *ccs_ptr* of the function is a pointer to the data interface calling structure, not to the control interface.

5.2.7 usb_class_cdc_get_ctrl_descriptor()

Searches for the descriptor of the control interface which controls the data interface identified by descriptor (intf_handle).

Synopsis

```
USB_STATUS usb_class_cdc_get_ctrl_descriptor
(
        _usb_device_instance_handle dev_handle,
        _usb_interface_descriptor_handle intf_handle,
        INTERFACE_DESCRIPTOR_PTR  * if_desc_ptr
)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

if_desc_ptr [in] — Pointer to control interface descriptor

Returns

USB_ OK (if found)

Traits

See also

INTERFACE_DESCRIPTOR_PTR

Description

This function is searching for the descriptor of the control interface which controls the data interface identified by the descriptor (intf_handle). Detected control interface descriptor is written to if_desc_ptr. This function must be called in the USB host stack locked state and validated USB device.

5.2.8 usb_class_cdc_get_ctrl_interface()

Find registered control interface in the chain.

Synopsis

```
CLASS_CALL_STRUCT_PTR usb_class_cdc_get_ctrl_interface
(
        void *intf_handle
)
```

Parameters

intf_handle [in] — Pointer to interface handle

Returns

Control interface instance

Traits

Description

This function is used to find registered control interface in the chain. The function must be called in the USB host stack locked state and validated USB device.

5.2.9 usb_class_cdc_get_data_interface()

Find registered data interface in the chain.

Synopsis

```
CLASS_CALL_STRUCT_PTR usb_class_cdc_get_data_interface
(
        void *intf_handle
)
```

Parameters

intf_handle [in] — Pointer to interface handle.

Returns

Data interface instance

Traits

Description

This function is used to find registered data interface in the chain. The function must be called in the USB host stack locked state and validated USB device.

5.2.10 usb_class_cdc_init_ipipe()

Start the interrupt endpoint to poll for an interrupt on a specified device.

Synopsis

```
USB_STATUS usb_class_cdc_init_ipipe
(
        CLASS_CALL_STRUCT_PTR acm_instance
)
```

Parameters

acm_instance [in] — ACM interface instance

Returns

Success as **USB_OK**

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

This function starts the interrupt endpoint to poll for an interrupt on a specified device.

5.2.11 usb_class_cdc_install_driver()

Add/install USB serial device driver.

Synopsis

```
USB_STATUS usb_class_cdc_install_driver
(
        CLASS_CALL_STRUCT_PTR data_instance,
        char * device_name
)
```

Parameters

data_instance [in] — Data instance

device_name [in] — Device name

Returns

Success as **USB_OK**

Traits

See also

usb_class_cdc_uninstall_driver(), **CLASS_CALL_STRUCT_PTR**

Description

This function adds/installs USB serial device driver into the MQX I/O subsystem. The following code string demonstrates the driver installation and its usage :

```
extern char string[STRING_MAX];
CLASS_CALL_STRUCT_PTR d_ccs;
const CDC_SERIAL_INIT init_struct = {
        USB_UART_NO_BLOCKING
};

/* get structure for communicating with CDC layer */
d_ccs = usb_class_cdc_get_data_interface(intf_handle);

if (d_ccs != NULL) {
        /* register MQX I/O device */
        if (USB_OK == usb_class_cdc_install_driver(intf, "ttyusb:")) {
                /* open instance of the device */
                FILE_PTR f = fopen("ttyusb:", (void*)&init_struct);

                if (f != NULL) {
                        int written;
                        /* write string to the opened device */
                        written = fwrite(string, 1, strlen(string), f);
                }
        }
}
```

5.2.12 usb_class_cdc_set_acm_ctrl_state()

Set RTS and DTR state of the line.

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_ctrl_state
(
        CLASS_CALL_STRUCT_PTR           ccs_ptr,
        uint8_t                         dtr,
        uint8_t                         rts
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

dtr [in] — DTR state to set (0 for active low, nonzero for active high)

rts [in]— RTS state to set (0 for active low, nonzero for active high)

Returns

Success as **USB_OK**

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

This function is used to set RTS and DTR state of the line.

NOTE

Although this function issues a command to the CDC control interface, the parameter *ccs_ptr* of the function is a pointer to the data interface calling structure, not to the control interface.

5.2.13 usb_class_cdc_set_acm_descriptors()

Set descriptors for the ACM interface.

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_descriptors
(
        CLASS_CALL_STRUCT_PTR ccs_ptr,
        USB_CDC_DESC_ACM_PTR acm_desc,
        USB_CDC_DESC_CM_PTR cm_desc,
        USB_CDC_DESC_HEADER_PTR header_desc,
        USB_CDC_DESC_UNION_PTR union_desc
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

acm_desc [in] — ACM functional descriptor pointer

cm_desc [in] — CM (call management) functional descriptor pointer

header_desc [in] — Header functional descriptor pointer

union_desc [in] — Union functional descriptor pointer

Returns

USB_OK (if validation passed)

Traits

See also

usb_class_cdc_get_acm_descriptors()

CLASS_CALL_STRUCT_PTR, **USB_HOST_DRIVER_INFO**,

USB_CDC_DESC_CM_PTR, **USB_CDC_DESC_HEADER_PTR**,

USB_CDC_DESC_UNION_PTR

Description

This function is used to set descriptors for the ACM interface. Descriptors can be used afterwards either by an application or by the driver.

5.2.14 usb_class_cdc_set_acm_line_coding()

Set parameters of the current line.

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_line_coding
(
        CLASS_CALL_STRUCT_PTR           ccs_ptr,
        USB_CDC_UART_CODING_PTR         uart_coding_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

uart_coding_ptr [in] — Location to store coding into

Returns

Success as **USB_OK**

Traits

See also

usb_class_cdc_get_acm_line_coding(), **CLASS_CALL_STRUCT_PTR**,

USB_CDC_UART_CODING_PTR

Description

This function is used to set parameters of the current line such as baud rate, HW control, etc.

NOTE

Although this function issues a command to the CDC control interface, the parameter *ccs_ptr* of the function is a pointer to the data interface calling structure, not to the control interface.

5.2.15 usb_class_cdc_unbind_acm_interface()

Unbind data interface from the appropriate control interface.

Synopsis

```
USB_STATUS usb_class_cdc_unbind_acm_interface
(
        CLASS_CALL_STRUCT_PTR           ccs_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

Returns

USB_OK

Traits

See also

usb_class_cdc_bind_acm_interface(), **CLASS_CALL_STRUCT_PTR**

Description

Data interface, which is specified by the ccs_ptr, will be unbound from the appropriate control interface. The function must be called in the USB host stack locked state and validated USB device.

5.2.16 usb_class_cdc_unbind_data_interfaces()

Unbind all data interfaces bound to the ACM control instance.

Synopsis

```
USB_STATUS usb_class_cdc_unbind_data_interfaces
(
        CLASS_CALL_STRUCT_PTR           ccs_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

Returns

USB_OK (if found)

Traits

See also

usb_class_cdc_bind_data_interfaces(), **CLASS_CALL_STRUCT_PTR**

Description

All data interfaces bound to the ACM control instance will be unbound from this interface. The function must be called in the USB host stack locked state and validated USB device.

5.2.17 usb_class_cdc_uninstall_driver()

Remove USB serial device driver.

Synopsis

```
USB_STATUS usb_class_cdc_uninstall_driver
(
        CLASS_CALL_STRUCT_PTR           data_instance
)
```

Parameters

data_instance [in] — Data instance.

Returns

Success as **USB_OK**

Traits

See also

usb_class_cdc_install_driver(), **CLASS_CALL_STRUCT_PTR**

Description

This function uninstalls the USB serial device driver from the MQX I/O subsystem.

5.3 HID Class API Function Listing

This section defines the API functions used for the Human interface Device (HID) class. The application can use these API functions to make HID applications by using a USB transport.

5.3.1 usb_class_hid_get_idle()

Read the idle rate of a particular HID device report.

Synopsis

```
USB_STATUS usb_class_hid_get_idle
(
        HID_COMMAND_PTR                 com_ptr,
        uint8_t                         rid,
        uint8_t *                       idle_rate
)
```

Parameters

com_ptr [in] — Class interface structure pointer

rid [in] — Report ID (see HID specification)

idle_rate [out] — Idle rate of this report

Returns

USB_OK (if command has been passed on USB)

Traits

See also

usb_class_hid_set_idle(), **HID_COMMAND_PTR**

Description

This function is called by the application to read the idle rate of a particular HID device report.

5.3.2 usb_class_hid_get_protocol()

Read the active protocol.

Synopsis

```
USB_STATUS usb_class_hid_get_protocol
(
        HID_COMMAND_PTR                 com_ptr,
        unsigned char                   *protocol
)
```

Parameters

com_ptr [in] — Class interface structure pointer

protocol [in] — Protocol (1 byte, 0 = Boot Protocol or 1 = Report Protocol)

Returns

USB_OK (if command has been passed on USB)

Traits

See also

usb_class_hid_set_protocol(), **HID_COMMAND_PTR**

Description

This function reads the active protocol, either a boot protocol or a report protocol.

5.3.3 usb_class_hid_get_report()

Get a report from the HID device.

Synopsis

```
USB_STATUS usb_class_hid_get_report
(
        HID_COMMAND_PTR                 com_ptr,
        uint8_t                         rid,
        uint8_t                         rtype,
        void                            *buf,
        uint16_t                        blen
)
```

Parameters

com_ptr [in] — Class interface structure pointer

rid [in] — Report ID (see HID specification)

rtype [in] — Report type (see HID specification)

buf [in] — Buffer to receive report data

blen [in] — Length of the Buffer

Returns

USB_OK (if command has been passed on USB)

Traits

See also

usb_class_hid_set_report(), **HID_COMMAND_PTR**

Description

This function is called by the application to get a report from the HID device.

5.3.4 usb_class_hid_init()

Initialize the class driver.

Synopsis

```
void usb_class_hid_init
(
        PIPE_BUNDLE_STRUCT_PTR                                  pbs_ptr,
        CLASS_CALL_STRUCT_PTR  ccs_ptr
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR, **PIPE_BUNDLE_STRUCT_PTR**

Description

This function is called by a common class to initialize the class driver. It is called in response to a select interface call by an application.

5.3.5 usb_class_hid_set_idle()

Silence a particular report on an interrupt in a pipe until a new event occurs or a specified time elapses.

Synopsis

```
USB_STATUS usb_class_hid_set_idle
(
        HID_COMMAND_PTR         com_ptr,
        uint8_t                 rid,
        uint8_t                 duration
)
```

Parameters

com_ptr [in] — Class interface structure pointer

rid [in] — Report ID (see HID specification)

duration [in] — Idle rate, 0 to inhibit regular reporting

Returns

USB_OK (if command has been passed on to USB)

Traits

See also

usb_class_hid_get_idle(), **HID_COMMAND_PTR**

Description

This function is called by the application to silence a particular report on an interrupt in a pipe until a new event occurs or a specified time elapses. The *duration* time is in units of 4-millisecond intervals, allowing a maximum of 1020 millisecond interval.

5.3.6 usb_class_hid_set_protocol()

Switch between the boot protocol and the report protocol, or vice versa.

Synopsis

```
USB_STATUS usb_class_hid_set_protocol
(
        HID_COMMAND_PTR         com_ptr,
        uint8_t                 protocol
)
```

Parameters

com_ptr [in] — Class interface structure pointer

protocol [in] — The protocol (0 = Boot, 1 = Report)

Returns

USB_OK (if command has been passed on USB)

Traits

See also

usb_class_hid_get_protocol(), **HID_COMMAND_PTR**

Description

This function switches between the boot protocol and the report protocol, or vice versa.

5.3.7 usb_class_hid_set_report()

Send a report to the HID device.

Synopsis

```
USB_STATUS usb_class_hid_set_report
(
        HID_COMMAND_PTR         com_ptr,
        uint8_t                 rid,
        uint8_t                 rtype,
        void                    *buf,
        uint16_t                blen
)
```

Parameters

com_ptr [in] — Class interface structure pointer

rid [in] — Report ID (see HID specification)

rtype [in] — Report type (see HID specification)

buf [in] — Buffer to receive report data

blen [in] — Length of the Buffer

Returns

USB_OK (if command has been passed on to USB)

Traits

See also

usb_class_hid_get_report(), **HID_COMMAND_PTR**

Description

This function is called by the application to send a report to the HID device.

5.4 MSD Class API Function Listing

This section defines the API functions used for the Mass Storage Class (MSD). The application can use these API functions to make MSD applications.

5.4.1 usb_class_mass_getmaxlun_bulkonly()

Get the Number of Logical Units on the device.

Synopsis

```
USB_STATUS usb_class_mass_getmaxlun_bulkonly
(
        CLASS_CALL_STRUCT_PTR  ccs_ptr,
        uint8_t                *pLUN,
        tr_callback            callback
        void                   *callback_param
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

pLUN [in] — Pointer to logical unit number variable

callback [in] — Callback upon completion

callback_param [in] — Parameter passed to the callback

Returns

ERROR STATUS (of the command)

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

This is a class specific command. See the documentation of the USB mass storage specifictaion to learn how this command works. This command is used the get the Number of Logical Units on the device. Caller will use the LUN number to direct the commands as a part of the CBW.

5.4.2 usb_class_mass_init()

Initialize the mass storage class.

Synopsis

```
void usb_class_mass_init
(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR  ccs_ptr
)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR, **PIPE_BUNDLE_STRUCT_PTR**

Description

This function initializes the mass storage class.

5.4.3 usb_class_mass_reset_recovery_on_usb()

Get a pending request from the class driver queue and send the RESET command to the control pipe.

Synopsis

```
USB_STATUS usb_class_mass_reset_recovery_on_usb
(
        USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr
)
```

Parameters

intf_ptr [in] — Interface structure pointer

Returns

ERROR STATUS (of the command)

Traits

See also

USB_MASS_CLASS_INTF_STRUCT_PTR

Description

This routine gets the pending request from the class driver queue and sends the RESET command to the control pipe. This routine is called when a phase of the pending command fails and class driver decides to reset the device. If there is no pending request in the queue, it will return. This routine registers a call back for control pipe commands to ensure that pending command is queued again.

NOTE

This functions should only be called by a callback or within a USB_lock() block.

5.4.4 usb_class_mass_storage_device_command()

Execute the command defined in the protocol API.

Synopsis

```
USB_STATUS usb_class_mass_storage_device_command
(
        CLASS_CALL_STRUCT_PTR  ccs_ptr,
        COMMAND_OBJECT_PTR     cmd_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

cmd_ptr [in] — Command

Returns

USB_OK — Means that command has been successfully queued in class driver queue (or has been passed to USB, if there is not other command pending).

Traits

See also

CLASS_CALL_STRUCT_PTR, **COMMAND_OBJECT_PTR**

Description

This routine is called by the protocol layer to execute the command defined in the protocol API. It can also be called directly by a user application if they wish to make their own, vendor-specific, commands to send to a mass storage device.

5.4.5 usb_class_mass_storage_device_command_cancel()

Dequeue the command in a class driver queue.

Synopsis

```
bool usb_class_mass_storage_device_command_cancel
(
        CLASS_CALL_STRUCT_PTR  ccs_ptr,
        COMMAND_OBJECT_PTR     cmd_ptr
)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

cmd_ptr [in] — Command.

Returns

- **ERROR STATUS** (error code)
- **USB_OK** — Means that a command has been successfully dequeued in a class driver queue.

Traits

See also

CLASS_CALL_STRUCT_PTR, **COMMAND_OBJECT_PTR**

Description

This function dequeues a command in a class driver queue.

5.4.6 usb_class_mass_cancelq()

Cancel the given request in the queue.

Synopsis

```
bool usb_class_mass_cancelq
(
        USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr,
        COMMAND_OBJECT_PTR     pCmd
)
```

Parameters

intf_ptr [in] — Interface structure pointer

pCmd [in] — Command object to be inserted in the queue

Returns

None

Traits

See also

COMMAND_OBJECT_PTR, **USB_MASS_CLASS_INTF_STRUCT_PTR**

Description

This routine cancels the given request in the queue.

5.4.7 usb_class_mass_deleteq()

Delete the pending request in the queue.

Synopsis

```
void usb_class_mass_deleteq
(
        USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr
)
```

Parameters

intf_ptr [in] — Interface structure pointer

Returns

None

Traits

See also

USB_MASS_CLASS_INTF_STRUCT_PTR

Description

This routine deletes the pending request in the queue.

5.4.8 usb_class_mass_get_pending_request()

Fetch the pointer to the first (pending) request in the queue, or NULL if there is no pending requests.

Synopsis

```
void usb_class_mass_get_pending_request
(
        USB_MASS_CLASS_INTF_STRUCT_PTR     intf_ptr,
        COMMAND_OBJECT_PTR  *              cmd *ptr
)
```

Parameters

intf_ptr [in] — Interface structure pointer

*cmd *ptr [in]* — Pointer to pointer which will hold the pending request

Returns

None

Traits

See also

HID_COMMAND_PTR, **USB_MASS_CLASS_INTF_STRUCT_PTR**

Description

This routine fetches the pointer to the first (pending) request in the queue, or NULL if there are no pending requests.

5.4.9 usb_class_mass_q_init()

Initialize a mass storage class queue.

Synopsis

```
void usb_class_mass_q_init
(
        USB_MASS_CLASS_INTF_STRUCT_PTR     intf_ptr
)
```

Parameters

intf_ptr [in] — Interface structure pointer

Returns

None

Traits

See also

USB_MASS_CLASS_INTF_STRUCT_PTR

Description

This function initializes a mass storage class queue.

5.4.10 usb_class_mass_q_insert()

Insert a command in the queue.

Synopsis

```
int32_t usb_class_mass_q_insert
(
        USB_MASS_CLASS_INTF_STRUCT_PTR                          intf_ptr,
        COMMAND_OBJECT_PTR              pCmd
)
```

Parameters

intf_ptr [in] — Interface structure pointer

pCmd [in] — Command object to be inserted in the queue

Returns

A position at which the insertion took place in the queue.

Traits

See also

COMMAND_OBJECT_PTR, **USB_MASS_CLASS_INTF_STRUCT_PTR**

Description

This function is called by the class driver to insert a command in the queue.

5.4.11 usb_mass_ufi_cancel()

Cancels the given request in the queue.

Synopsis

```
bool usb_mass_ufi_cancel
(
        COMMAND_OBJECT_PTR     cmd_ptr
)
```

Parameters

cmd_ptr [in] — Command object pointer

Returns

None

Traits

See also

COMMAND_OBJECT_PTR

Description

This function cancels the given request in the queue.

5.4.12 usb_mass_ufi_generic()

Initialize the mass storage class.

Synopsis

```
USB_STATUS usb_mass_ufi_generic
(
        /* [in] command object allocated by application*/
        COMMAND_OBJECT_PTR     cmd_ptr,
        uint8_t                opcode,
        uint8_t                lun,
        uint32_t               lbaddr,
        uint32_t               blen,
        uint8_t                cbwflags,
        unsigned char          *buf,
        uint32_t               buf_len
)
```

Parameters

cmd_ptr [in] — Command object pointer

opcode [in] — Opcode of command block

lun [in] — Logical unit number of command block

lbaddr [in] — Logical block address

blen [in] — Allocation length

cbwflags [in] — Command block wrapper flags

buf [in] — Command data buffer

buf_len [in] — Command data buffer length

Returns

None

Traits

See also

COMMAND_OBJECT_PTR

Description

This function initializes the mass storage class.

5.5 PHDC Class API Function Listing

This section defines the API functions used for the Personal Healthcare (PHDC) class. The application can use these API functions to make PHDC applications using the USB transport.

5.5.1 usb_class_phdc_init()

Initialize the PHDC interface structure with the attached device specific information containing descriptors and communication pipe handles.

Synopsis

```
void usb_class_phdc_init
(
        PIPE_BUNDLE_STRUCT_PTR    pbs_ptr,
        CLASS_CALL_STRUCT_PTR     ccs_ptr
)
```

Parameters

pbs_ptr [in] — Pointer to the pipe bundle structure containing USB pipe information for the attached device

ccs_ptr [in] — PHDC call structure pointer. This structure contains a class validity-check code and a pointer to the current interface handle.

Returns

None

Traits

See also

CLASS_CALL_STRUCT_PTR,

PIPE_BUNDLE_STRUCT_PTR

Description

This function serves the main purpose of initializing the PHDC interface structure with the attached device specific information containing descriptors and communication pipe handles.

The **usb_class_phdc_init()** function is usually called by the common-class layer services as the result of an interface select function call from the Application / IEEE 11073 Manager. The application will select the interface after receiving the **USB_ATTACH** indication event from the USB host API.

5.5.2 usb_class_phdc_set_callbacks()

Register the application defined callback functions for the PHDC send, receive, and control request actions.

Synopsis

```
USB_STATUS usb_class_phdc_set_callbacks
(
        CLASS_CALL_STRUCT_PTR     ccs_ptr,
        phdc_callback             sendCallback,
        phdc_callback             recvCallback,
        phdc_callback             ctrlCallback
)
```

Parameters

ccs_ptr [in] — Pointer to the current PHDC interface instance for which the callbacks are set

sendCallback [in] — Function pointer for the send Callback function

recvCallback [in] — Function pointer for the receive Callback function

ctrlCallback [in] —Function pointer for the send Control Callback function

Returns

- **USB_OK** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_TRANSFER_IN_PROGRESS** (As there are still pending transfers on the data pipes, the request to register the callbacks was denied. No previously registered callback was affected)

Traits

See also

CLASS_CALL_STRUCT_PTR

Description

The usb_class_phdc_set_callbacks() function is used to register the application defined callback functions for the PHDC send, receive, and control request actions. Providing a non-NULL pointer to a callback function, phdc_callback type, will register the provided function to be called when the corresponding action is completed. Additionally, providing a NULL pointer will invalidate the callback for the corresponding action.

Application registered callbacks are unique for each selected PHDC interface. Only one send callback and one receive callback can be registered for each PHDC interface. Since the PHDC class supports multiple send/receive actions to be queued in the lower layers at the same time, the application can identify the action for which the callback function was called by using the call_param pointer that can point to a different location for each Send/Receive/Ctrl function call. The call_param pointer is transmitted as parameter to the PHDC Send/Receive/Ctrl functions and it is returned to the application when the Send/Receive/Ctrl callback function is called. Before saving the callback pointers in the PHDC interface structure, the usb_class_phdc_set_callbacks() function verifies all the transfer pipes for pending transactions. The callbacks for send/receive actions cannot be changed while transactions are pending on

the pipes. In this case, the function will deny the set callbacks request and will return **USBERR_TRANSFER_IN_PROGRESS**.

If the pipes have no pending transactions, the **usb_class_phdc_set_callbacks()**() function will save the callbacks pointers in the current interface structure and will return **USB_OK**.

At USB transfer completion, the user registered callbacks, such as sendCallback, recvCallback, or controlCallback, will be called from the PHDC class after the internal processing of the transfer status and using the provided callback_param at the action start.

5.5.3 usb_class_phdc_send_control_request()

Set feature of specified hub port.

Synopsis

```
USB_STATUS usb_class_phdc_send_control_request
(
    USB_PHDC_PARAM *call_param
)
```

Parameters

call_param [in] — Pointer to a USB_PHDC_PARAM structure

Returns

- **USB_OK / USB_STATUS_TRANSFER_QUEUED** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_ERROR** (parameter error)
- **USBERR_INVALID_REQ_TYPE** (invalid type for the request)
- **USBERR_TRANSFER_IN_PROGRESS** a control request **SET / CLEAR_FEATURE** is already in progress

Traits

See also

USB_PHDC_PARAM

Description

The usb_class_phdc_send_control_request() function is used to send PHDC class-specific request to the attached device. As defined by the PHDC class specification, the request must be one of the following types: **SET_FEATURE, CLEAR_FEATURE, GET_STATUS.**

SET_FEATURE, **CLEAR_FEATURE** requests**:** In order not to stall the device endpoint, the usb_class_phdc_send_control_request() function will, first, verify if the attached device supports metadata preamble transfer feature for the **SET_FEATURE** and **CLEAR_FEATURE** request. If the preamble capability is not supported, this function will return **USBERR_INVALID_REQ_TYPE** and exit. Only one **SET_FEATURE**/**CLEAR_FEATURE** control request to the device can be queued on the control pipe at a time. In case there is another request pending, this function will deny the request by returning **USBERR_TRANSFER_IN_PROGRESS**. Additionally, for the **SET_FEATURE** and **CLEAR_FEATURE** requests, this function will verify the pending transfers on the data pipes. To avoid synchronization issues with the preamble, the PHDC will not transmit the control request if the data pipes have transfers queued for the device. In this case, the function will return **USBERR_TRANSFER_IN_PROGRESS** and exit. The application is also responsible for checking the device endpoint by issuing a **GET_STATUS** request before sending a **SET_FEATURE** or **CLEAR_FEATURE** to the device.

GET_STATUS requests: There are no restrictions in terms of pending requests on the control pipe as the GET_STATUS request will neither interfere with the other PHDC send/receive functions nor will it cause sync issues on the device.

PHDC Send Control Callback: The completion of the PHDC control request is managed internally by the PHDC class for also handling the device endpoint stall situation. If the PHDC is informed by the USB host API that the device control endpoint is stalled, the PHDC will attempt to clear the endpoint STALL by issuing a standard **CLEAR_FEATURE** command request to the device. In the end, the PHDC calls the application registered callback for the control request function by using the USB provided status code and the PHDC class status code (through the call_param >usb_status pointer). If the PHDC fails to clear the endpoint stall, it will call the application send control callback with the **USB_PHDC_ERR_ENDP_CLEAR_STALL** PHDC status.

5.5.4 usb_class_phdc_recv_data()

Receive PHDC class specific data or metadata packets.

Synopsis

```
USB_STATUS usb_class_phdc_recv_data
(
    USB_PHDC_PARAM *call_param
)
```

Parameters

call_param [in] — Pointer to a **USB_PHDC_PARAM** structure

Returns

- **USB_OK / USB_STATUS_TRANSFER_QUEUED** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_ERROR** (parameter error)
- **USBERR_TRANSFER_IN_PROGRESS** a control request **SET / CLEAR_FEATURE** is in progress

Traits

See also

USB_PHDC_PARAM

Description

The **usb_class_audio_recv_data()** function is used for receiving PHDC class specific data or metadata packets. It schedules a USB receive on the QoS — selected pipe for the lower host API. The receive transfer will end either when the host has received the specified amount of bytes, or if the last packet received is less than the pipe maximum packet size (**MAX_PACKET_SIZE**) indicating that the device does not have more data to send. Before scheduling the receive action, this function will first validate the provided call_param pointer and Rx relevant fields by checking the call_param->ccs_ptr (class interface), call_param->qos (QoS bitmap used to identify the pipe for receive), the call_param->buff_ptr (buffer for storing the data received — cannot be NULL), and call_param->buff_size (number of bytes to receive — cannot be 0). If all the parameters are valid, the function checks if a **SET_FEATURE** or **CLEAR_FEATURE** control request is pending. If either is pending, the function returns **USBERR_TRANSFER_IN_PROGRESS** and the transaction is refused. The PHDC does not know if the device has metadata feature enabled to decode the received packet.

NOTE

To prevent memory alignment issues on certain platforms, it is recommended that the provided receive size, call_param->buff_size, always be a multiple of 4 bytes.

If all checks are passing, this function initiates a USB host receive action on the designated pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Receive Callback:

The PHDC internal Receive Callback will be called when the USB Host API reception completes. The callback will parse the received data, populate the PHDC status codes in the **USB_PHDC_PARAM** structure and call the user defined receive callback (the function registered by the user using the **usb_class_phdc_set_callbacks()**).

The parameters passed to the user registered callback are:

- **USB_PHDC_PARAM** structure.
  - Through usb_phdc_status, this structure will inform the user if data received are metadata preamble or regular data and if metadata preamble or regular data were expected.
  - Through usb_status, this structure will inform the user callback about the status of the USB transfer.

The PHDC receive callback also checks whether the data recevied are plain data or metadata and compares the result with the type of data that was expected. If the host was expecting a metadata but only plain data was received, according to the health care standard, the host will issue a **SET_FEATURE** (**ENDPOINT_HALT**) followed by a **CLEAR_FEATURE (ENDPOINT_HALT)** on the receiving pipe.

5.5.5 usb_class_phdc_send_data()

Send PHDC class specific data or metadata packets.

Synopsis

```
USB_STATUS usb_class_phdc_send_data
(
    USB_PHDC_PARAM                                          *call_param
)
```

Parameters

call_param [in] — Pointer to a USB_PHDC_PARAM structure

Returns

- **USB_OK / USB_STATUS_TRANSFER_QUEUED** (success)
- **USBERR_NO_INTERFACE** (the provided interface is not valid)
- **USBERR_INVALID_BMREQ_TYPE** (invalid qos bitmap fields in the sending packet)
- **USBERR_ERROR** (parameter error / metadata checking error)
- **USBERR_TRANSFER_IN_PROGRESS** a control request **SET / CLEAR_FEATURE** is in progress

See also

USB_PHDC_PARAM

Description

The usb_class_phdc_send_data function is used for sending PHDC class specific data or metadata packets. It schedules a USB send transfer on the bulk-out pipe for the lower host API. Before scheduling the send action, this function will first validate the provided call_param pointer and Tx relevant fields by checking the call_param->ccs_ptr (class interface), the call_param->buff_ptr (buffer for taking the data to be sent- cannot be NULL), and call_param->buff_size (number of bytes to send - cannot be 0). If the parameters are valid, this function validates the data buffer provided by the application for transmission. The usb_class_phdc_send function expects that the application will provide the data buffer constructed accordingly with the metadata preamble feature. The application is responsible for forming the data packet to be sent including the metadata preamble, **USB_PHDC_METADATA_PREAMBLE**, if it is used.

If metadata is included in the packet, call_param_ptr->metadata is TRUE, the attached device supports metadata and the metadata feature was already set on the device by using the **usb_class_phdc_send_control_request()** function. This function will then validate the QoS in the transmit packet by checking its bitmap fields and using the QoS descriptor for the PHDC Bulk-Out pipe. If the requested QoS is not supported in the descriptor, this function denies the transfer and returns **USBERR_ERROR**.

Before sending the data, this function also checks if there are pending **SET /CLEAR_FEATURE** request types to the device. Until those are completed, the send function does not know if the device has the metadata preamble feature activated. Therefore, it will deny the requested transfer and will return **USBERR_TRANSFER_IN_PROGRESS**. If all the checks are passing, this function initiates a USB

host send action on the Bulk-Out pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Send Callback:

The PHDC internal Send Callback will be called when the USB host API send transfer completes. The callback will populate the PHDC status codes in the **USB_PHDC_PARAM** structure and call the user defined receive callback (the function registered by the user using the usb_class_phdc_set_callbacks). The parameters passed to the user registered callback are:

- **USB_PHDC_PARAM** structure
  - The usb_phdc_status is set to **USB_PHDC_TX_OK** when the received status code from USB host API is **USB_OK**, or **USB_PHDC_ERR** otherwise.
  - Through the usb_status, this structure pointer informs the user callback about the status of the USB transfer.

The device endpoint stall situation is also handled by the internal send callback. If the PHDC is informed by the USB host API that the device endpoint is stalled, the PHDC will attempt to clear the endpoint STALL by issuing a standard **CLEAR_FEATURE** command request to the device. If the PHDC fails to clear the endpoint stall, it will call the application send control callback with the **USB_PHDC_ERR_ENDP_CLEAR_STALL** PHDC status.

5.5.6 usb_class_phdc_cancel_transfer()

Attempt to cancel the indicated transfer.

Synopsis

```
USB_STATUS usb_class_phdc_cancel_transfer
(
CLASS_CALL_STRUCT_PTR ccs_ptr,
uint32_t               tr_index,
_usb_pipe_handle      tr_pipe_handle
)
```

Parameters

ccs_ptr [in] — Pointer to the current PHDC interface instance for which the callbacks are set

tr_index [in] Specific transfer to cancel

tr_pipe_handle [in] Pipe handle

Returns

- **USB_OK** (success)
- Others (failure)

Description

The function is used to attempt to cancelling the indicated transfer.

Chapter 6 Data Structures

6.1 USB Host Layer Structures

6.1.1 INTERFACE_DESCRIPTOR_PTR

The Communications Interface Class uses the standard interface descriptor as defined in Chapter 9 of the *USB Specification*.

Synopsis

```
typedef struct usb_interface_descriptor
{
        uint8_t                 bLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bInterfaceNumber;
        uint8_t                 bAlternateSetting;
        uint8_t                 bNumEndpoints;
        uint8_t                 bInterfaceClass;
        uint8_t                 bInterfaceSubClass;
        uint8_t                 bInterfaceProtocol;
        uint8_t                 iInterface;
} INTERFACE_DESCRIPTOR,  * INTERFACE_DESCRIPTOR_PTR;
```

Fields

bLength — Descriptor size in bytes = 9.

bDescriptorType — INTERFACE descriptor type = 4.

bInterfaceNumber — Interface number.

bAlternateSetting — Value to select this IF.

bNumEndpoints — Number of endpoints excluding 0.

bInterfaceClass — Class code, 0xFF = vendor specific.

bInterfaceSubClass — Sub-Class code, 0 if class = 0.

bInterfaceProtocol — Protocol, 0xFF = vendor specific.

iInterface — Index to interface string.

6.1.2 PIPE_BUNDLE_STRUCT_PTR

Pipe bundle = device handle + interface handle + 1..N pipe handles.

NOTE

The pipe handles are for non-control pipes only, i.e. pipes belonging strictly to this interface. The control pipe belongs to the device even if it is being used by device interfaces. Hence, a pointer to the device instance is provided. Closing pipes for the interface does NOT close the control pipe which may still be required to set new configurations/interfaces etc.

Synopsis

```
typedef struct pipe_bundle_struct
```

```
{
        _usb_device_instance_handle          dev_handle;
        _usb_interface_descriptor_handle     intf_handle;
        _usb_pipe_handle                     pipe_handle[4];
} PIPE_BUNDLE_STRUCT,  * PIPE_BUNDLE_STRUCT_PTR;
```

Fields

dev_handle — device handle

intf_handle — interface handle

pipe_handle[4] — pipe handle

6.1.3 PIPE_INIT_PARAM_STRUCT

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

Synopsis

```
typedef struct
{
        void                    *DEV_INSTANCE;
        uint32_t                INTERVAL;
        uint32_t                MAX_PACKET_SIZE;
        uint32_t                NAK_COUNT;
        uint32_t                FIRST_FRAME;
        uint32_t                FIRST_UFRAME;
        uint32_t                FLAGS;
        uint8_t                 DEVICE_ADDRESS;
        uint8_t                 ENDPOINT_NUMBER;
        uint8_t                 DIRECTION;
        uint8_t                 PIPETYPE;
        uint8_t                 SPEED;
        uint8_t                 TRS_PER_UFRAME;
} PIPE_INIT_PARAM_STRUCT,  * 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 the following:

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 the following:

USB_RECV

USB_SEND

PIPE_TYPE — Type of transfer to make on the pipe. One of the following:

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 the following:

1 (default): one transaction

2 two transactions

3 three transactions

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

6.1.4 TR_INIT_PARAM_STRUCT

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

Synopsis

```
typedef struct
{
        uint32_t                TR_INDEX;
        unsigned char           *TX_BUFFER;
        unsigned char           *RX_BUFFER;
        uint32_t                TX_LENGTH;
        uint32_t                RX_LENGTH;
        tr_callback             CALLBACK;
        void                    *CALLBACK_PARAM;
        unsigned char           *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.

6.1.5 USB_HOST_DRIVER_INFO

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

Synopsis

```
typedef struct driver_info
{
        uint8_t                 IDVENDOR[2];
        uint8_t                 IDPRODUCT[2];
        uint8_t                 BDEVICECLASS;
        uint8_t                 BDEVICESUBCLASS;
        uint8_t                 BDEVICEPROTOCOL;
        uint8_t                 RESERVED;
        event_callback          ATTACH_CALL;
} USB_HOST_DRIVER_INFO,  * 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.

6.2 Common class structures

6.2.1 CLASS_CALL_STRUCT_PTR

This structure is for storing a class validity-check code with the pointer to the data. The address of one such structure is passed as a pointer to select-interface calls where values for that interface get initialized. Then, the structure should be passed to class calls by using the interface.

Synopsis

```
typedef struct class_call_struct
{
        _usb_class_intf_handle    class_intf_handle;
        uint32_t                  code_key;
        void                      *next;
        void                      *anchor;
}CLASS_CALL_STRUCT,  * CLASS_CALL_STRUCT_PTR;
```

Fields

class_intf_handle — Class interface handle.

code_key — Code key.

next — Pointer to the next CLASS_CALL_STRUCT.

anchor — Pointer to the first CLASS_CALL_STRUCT.

6.2.2 GENERAL_CLASS

This structure is used for storing generic information for every class driver instance. Every class instance should embed this structure at the beginning of the class instance structure allowing the USB stack to browse the currently attached devices.

Synopsis

```
typedef struct general_class
{
        struct general_class  *  next;
        struct general_class  *  anchor;
        _usb_device_instance_handle dev_handle;
        _usb_host_handle            host_handle;
        uint32_t                     key_code;
}CLASS_CALL_STRUCT,  * CLASS_CALL_STRUCT_PTR;
```

Fields

next — Pointer to the next class instance.

annchor — Pointer to the first class instance.

dev_handle — Device handle attached to the class instance.

host_handle — Host handle which manages the class instance.

key_code — Internal key assigned by USB stack used to validate the instance.

6.3 Audio class structures

6.3.1 AUDIO_COMMAND_PTR

The Audio command structure.

Synopsis

```
typedef struct
{
        CLASS_CALL_STRUCT_PTR     CLASS_PTR;
        tr_callback               CALLBACK_FN;
        void                      *CALLBACK_PARAM;
} AUDIO_COMMAND,  * AUDIO_COMMAND_PTR;
```

Fields

CLASS_PTR — Pointer to class call structure.

CALLBACK_FN — Callback function.

CALLBACK_PARAM — Callback function parameter.

6.3.2 USB_AUDIO_CTRL_DESC_HEADER_PTR

The class-specific descriptor starts with a header. The bcdCDC field identifies the release of the USB Class Definitions for Audio Devices Specification with which this interface and its descriptors comply.

Synopsis

```
typedef struct
{
        uint8_t                   bFunctionLength;
        uint8_t                   bDescriptorType;
        uint8_t                   bcdCDC[2];
        uint8_t                   wTotalLength[2];
        uint8_t                   bInCollection;
} USB_AUDIO_DESC_HEADER,  * USB_AUDIO_DESC_HEADER_PTR;
```

Fields

bFunctionLength — Size of the descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — Header functional descriptor subtype as defined in [USBCDC 1.2].

bcdCDC[2] — Release number of [USBCDC 1.2] in BCD, with implies a decimal point between bits 7 and 8 (0x0120=1.20-1.2).

wTotalLength — Total number of bytes returned for the class-specific AudioControl interface descriptor.

bInCollection — The number of AudioStreaming and MIDIStream interfaces in the Audio interface Collection to which this AudioControl interface belongs.

6.3.3 USB_AUDIO_CTRL_DESC_IT_PTR

Input Terminal Descriptor structure.

Synopsis

```
typedef struct
{
        uint8_t                 bFunctionLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bTerminalID;
        uint8_t                 wTerminalType[2];
        uint8_t                 bAssocTerminal;
        uint8_t                 bNrChannels;
        uint8_t                 wChannelCofig[2];
        uint8_t                 iChannelNames;
        uint8_t                 iTerminal;
} USB_AUDIO_CTRL_DESC_IT,  * USB_AUDIO_CTRL_DESC_IT_PTR;
```

Fields

bFunctionLength — Size of this descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — INPUT_TERMINAL.

bTerminalID — Constant which uniquely identifies the Terminal within the audio function.

wTerminalType — Constant which characterizes the type of the Terminal.

bAssocTerminal — ID of the Output Terminal with which this Input Terminal is associated.

bNrChannels — A number of logical output channels in the output audio channel cluster of the Terminal.

wChannelCofig — Describes the spatial location of the logical channels.

iChannelNames — Index of a string descriptor which describes the name of the first logical channel.

iTerminal — Index of a string descriptor which describes the Input Terminal.

6.3.4 USB_AUDIO_CTRL_DESC_OT_PTR

Output Terminal Descriptor structure.

Synopsis

```
typedef struct
{
        uint8_t                 bFunctionLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bTerminalID;
        uint8_t                 wTerminalType[2];
        uint8_t                 bAssocTerminal;
        uint8_t                 bSourceID;
        uint8_t                 iTerminal;
} USB_AUDIO_CTRL_DESC_OT,  * USB_AUDIO_CTRL_DESC_OT_PTR;
```

Fields

bFunctionLength — Size of the descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — OUTPUT_TERMINAL.

bTerminalID — Constant which uniquely identifies the Terminal within the audio function.

wTerminalType — Constant which characterizes type of the Terminal.

bAssocTerminal — ID of the Input Terminal with which the Output Terminal is associated.

bSourceID — ID of the Unit or Terminal to which this Terminal is connected.

iTerminal — Index of a string descriptor which describes the Input Terminal.

6.3.5 USB_AUDIO_CTRL_DESC_FU_PTR

Pointer to a Feature Unit Descriptor structure.

Synopsis

```
typedef struct
{
        uint8_t                 bLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bUnitID;
        uint8_t                 bSourceID;
        uint8_t                 bControlSize;
        uint8_t                 bmaControls[];
} USB_AUDIO_CTRL_DESC_FU,  * USB_AUDIO_CTRL_DESC_FU_PTR;
```

Fields

bFunctionLength — Size of the descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — FEATURE_UNIT.

bUnitID — Constant which uniquely identifies the Unit within the audio function.

bSourceID — ID of the Unit or Terminal to which this Feature Unit is connected.

bControlSize — Size, in bytes, of an element of the bmaControls array.

6.3.6 USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR

Pointer to a Class-specific Audio stream interface descriptor.

Synopsis

```
typedef struct
{
        uint8_t                 bLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bTerminalLink;
        uint8_t                 bDelay;
        uint8_t                 bFormatTag[2];
} USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF,
 *  USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR;
```

Fields

bLength — Size of the descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — AS_GENERAL.

bTerminalLink —The Terminal ID of the Terminal to which the endpoint of this interface is connected.

bDelay — Introduced by the data path.

wFormatTag — The Audio Data Format that has to be used to communicate with this interface.

6.3.7 USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR

Pointer to a format type descriptor.

Synopsis

```
typedef struct
{
        uint8_t                 bLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bFormatType;
        uint8_t                 bNrChannels;
        uint8_t                 bSubFrameSize;
        uint8_t                 bBitResolution;
        uint8_t                 bSamFreqType;
        uint8_t                 bSamFreq[3];
} USB_AUDIO_STREAM_DESC_FORMAT_TYPE,
 * USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR;
```

Fields

bLength — Size of the descriptor.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — FORMAT_TYPE.

bFormatType — Constant which identifies the Format Type that the Audio Stream interface is using.

bNrChannels — Indicates the number of physical channels in the audio data stream.

bSubFrameSize — The number of bytes occupied by one audio subframe. The number of bytes can be 1, 2, 3, or 4.

bBitResolution — The number of effectively used bits from the available bits in an audio subframe.

bSamFreqType — Indicates how the sampling frequency can be programmed.

bSamFreq[3] — Sampling frequency in Hz for the isochronous data endpoint.

6.3.8 USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

Pointer to a Class-specific Isochronous Audio Data Endpoint descriptor.

Synopsis

```
typedef struct
{
        uint8_t                 bLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubType;
        uint8_t                 bmAttributes;
        uint8_t                 bLockDelayUnits;
        uint8_t                 wLockDelay[2];
} USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP,
 * USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR;
```

Fields

bLength — Size of the descriptor in bytes.

bDescriptorType — CS_ENDPOINT.

bDescriptorSubtype — EP_GENERAL.

bmAttributes — A bit in the range D6..0 set to 1 indicates that the mentioned control is supported by this endpoint.

bLockDelayUnits — Indicates the units used for the wLockDelay field (0 for undefined, 1 for milliseconds, 2 for PCM sample time).

wLockDelay — Indicates the time it takes this endpoint to reliably lock its internal clock recovery circuitry. Units used depend on the value of the bLockDelayUnits field.

6.4 CDC class structures

6.4.1 USB_CDC_DESC_ACM_PTR

Abstract control management functional descriptor.

Synopsis

```
typedef struct
{
        uint8_t                 bFunctionLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubtype;
        #define USB_ACM_CAP_COMM_FEATURE 0x01
        #define USB_ACM_CAP_LINE_CODING 0x02
        #define USB_ACM_CAP_SEND_BREAK 0x04
        #define USB_ACM_CAP_NET_NOTIFY 0x08
        uint8_t                 bmCapabilities;
} USB_CDC_DESC_ACM,  * USB_CDC_DESC_ACM_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — Abstract control management functional descriptor subtype as defined in [USBCDC1.2].

bmCapabilities — Specifies the capabilities that this data/fax function supports. A bit value of zero means that the capability is not supported.

D7..D4: RESERVED (Reset to zero).
D3: Function generates the notification NetworkConnect ION.
D2: Function supports the management element SendBreak.
D1: Function supports the management elements GetLineCoding, SetControlLineState, GetLineCoding. Function will generate the notification SerialState.
D0: Function supports management elements GetCommFeature, SetCommFeature and ClearCommFeature.

6.4.2 USB_CDC_DESC_CM_PTR

Call management functional descriptor.

Synopsis

```
typedef struct
{
        uint8_t                   bFunctionLength;
        uint8_t                   bDescriptorType;
        uint8_t                   bDescriptorSubtype;
        #define USB_ACM_CM_CAP_HANDLE_MANAGEMENT     0x01
        #define USB_ACM_CM_CAP_DATA_CLASS            0x02
        uint8_t                   bmCapabilities;
        uint8_t                   bDataInterface;
} USB_CDC_DESC_CM,  * USB_CDC_DESC_CM_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — Call management functional descriptor subtype as defined in [USBCDC1.2].

bmCapabilities — Specifies the capabilities that this data/fax function supports. A bit value of zero means that the capability is not supported.

D7..D2: RESERVED (Reset to zero).

D1: 0 - Function sends/receives call management information only over this Communications Class interface

1 – Function can send/receive call management information over the Data Class interface.

D0: 0 – Function does not perform call management
1 – Function does perform call management

bDataInterface — bInterfaceNumber of the Data Class interface.

6.4.3 USB_CDC_DESC_HEADER_PTR

The class-specific descriptor starts with a header. The *bcdCDC* field identifies the release of the *USB Class Definitions for Communications Devices Specification* with which this interface and its descriptors comply.

Synopsis

```
typedef struct
{
        uint8_t                 bFunctionLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubtype;
        uint8_t                 bcdCDC[2];
} USB_CDC_DESC_HEADER,  * USB_CDC_DESC_HEADER_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — Header functional descriptor subtype as defined in [USBCDC1.2].

bcdCDC[2] — Release number of [USBCDC1.2] in BCD, with implied decimal point between bits 7 and 8 (0x0120=1.20=1.2)

6.4.4 USB_CDC_DESC_UNION_PTR

Union Functional Descriptor describes the relationship between a group of interfaces that can be considered to form a functional unit. It can only occur within the class-specific portion of an Interface descriptor. One of the interfaces in the group is designated to be a master or a controlling interface for the other (e.g. data) interface. Notifications for the entire group can be sent from this interface, but apply to the entire group of interfaces. Interfaces in this group can include Communications, Data, or any other valid USB interface class including, but not limited to, Audio, HID, and Monitor.

Synopsis

```
typedef struct
{
        uint8_t                 bFunctionLength;
        uint8_t                 bDescriptorType;
        uint8_t                 bDescriptorSubtype;
        uint8_t                 bMasterInterface;
        uint8_t                 bSlaveInterface[];
} USB_CDC_DESC_UNION,  * USB_CDC_DESC_UNION_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes.

bDescriptorType — CS_INTERFACE.

bDescriptorSubtype — Union functional descriptor subtype as defined in [USBCDC1.2].

bMasterInterface — The interface number of this ACM interface.

bSlaveInterface — The interface number of the Data Class interface.

6.4.5 USB_CDC_UART_CODING_PTR

This structure configures the UART.

Synopsis

```
typedef struct {
    uint32_t      baudrate;
    uint8_t       stopbits;
    uint8_t       parity;
    uint8_t       databits;
} USB_CDC_UART_CODING,  * USB_CDC_UART_CODING_PTR;
```

Fields

baudrate — Baud rate.

stopbits — Stop bits (1 ~ 1bit, 2 ~ 2bits, 3 ~ 1.5bit).

parity — Parity (1 ~ even, -1 ~ odd, 0 ~ no parity).

databits — Data bits.

6.5 HID class structures

6.5.1 HID_COMMAND_PTR

The HID command structure.

Synopsis

```
typedef struct
{
        CLASS_CALL_STRUCT_PTR     CLASS_PTR;
        tr_callback               CALLBACK_FN;
        void                      *CALLBACK_PARAM;
} HID_COMMAND,  * HID_COMMAND_PTR;
```

Fields

CLASS_PTR — Pointer to class call structure.

CALLBACK_FN — Callback function.

CALLBACK_PARAM — Callback function parameter.

6.6 MSD class structures

6.6.1 COMMAND_OBJECT_PTR

This function is used for MSD class. There is one single command object for all protocols.

Synopsis

```
typedef struct _COMMAND_OBJECT
{
        CLASS_CALL_STRUCT_PTR     CALL_PTR;
        uint32_t                  LUN;
        CBW_STRUCT_PTR            CBW_PTR;
        CSW_STRUCT_PTR            CSW_PTR;
```

```
        void (_CODE_PTR CALLBACK)      (USB_STATUS,void*,void*,uint32_t);
        void                        *DATA_BUFFER;
        uint32_t                    BUFFER_LEN;
        USB_CLASS_MASS_COMMAND_STATUS STATUS;
        USB_CLASS_MASS_COMMAND_STATUS PREV_STATUS;
        uint32_t                    TR_BUF_LEN;
        uint8_t                     RETRY_COUNT;
        uint8_t                     TR_INDEX;
} COMMAND_OBJECT_STRUCT,  * COMMAND_OBJECT_PTR;
```

Fields

CALL_PTR — Class intf data pointer and key.

LUN — Logical unit number on device.

CBW_PTR — Current CBW being constructed.

CSW_PTR — CSW for this command.

CALLBACK — Command callback.

USB_STATUS — Status of this command.

*void** — Pointer to USB_MASS_BULK_ONLY_REQUEST_STRUCT.

*void** — Pointer to the command object.

unit32_t — Length of the data transferred if any.

DATA_BUFFER — Buffer for IN/OUT for the command.

BUFFER_LEN — Length of data buffer.

STATUS — Current status of this command.

PREV_STATUS — Previous status of this command.

TR_BUF_LEN —Length of the buffer received in currently executed TR.

RETRY_COUNT — Number of times this command tried.

TR_INDEX — TR_INDEX of the TR used for search.

6.6.2 USB_MASS_CLASS_INTF_STRUCT_PTR

USB Mass Class Interface structure. This structure will be passed to all commands to this class driver. The structure holds all information pertaining to an interface on the storage device. This allows the class driver to know which interface the command is directed for.

Synopsis

```
typedef struct _Usb_Mass_Intf_Struct
{
        GENERAL_CLASS             G;
        _usb_pipe_handle          CONTROL_PIPE;
        _usb_pipe_handle          BULK_IN_PIPE;
        _usb_pipe_handle          BULK_OUT_PIPE;
        MASS_QUEUE_STRUCT         QUEUE;
        uint8_t                   INTERFACE_NUM;
        uint8_t                   ALTERNATE_SETTING;
} USB_MASS_CLASS_INTF_STRUCT,  * USB_MASS_CLASS_INTF_STRUCT_PTR;
```

Fields

G — Embedded generic class data.

CONTROL_PIPE — Control pipe handle.

BULK_IN_PIPE — Bulk in pipe handle.

BULK_OUT_PIPE — Bulk out pipe handle.

QUEUE — Structure that queues requests.

INTERFACE_NUM — Interface number.

ALTERNATE_SETTING — Alternate setting.

6.7 PHDC class structures

6.7.1 USB_PHDC_PARAM

PHDC required type for the parameter passing to the PHDC transfer functions (Send / Receive/ Ctrl). A pointer to this type is required when those functions are called. The same pointer will also be transmitted back to the application when the corresponding callback function is called by the PHDC by using the callback_param_ptr.

The application can maintain a linked list of transfer request pointers, knowing, at any moment, what the pending transactions with the PHDC are.

Synopsis

```
typedef struct usb_phdc_param_type
{
   CLASS_CALL_STRUCT_PTR ccs_ptr;
   uint8_t classRequestType;
   bool metadata;
   uint8_t qos;
   uint8_t* buff_ptr;
   uint32_t buff_size;
   uint32_t tr_index;
   _usb_pipe_handle tr_pipe_handle;
   uint8_t usb_status;
   uint8_t usb_phdc_status;
} USB_PHDC_PARAM;
```

Fields

ccs_ptr — Pointer to **CLASS_CALL_STRUCT** which identifies the interface.

class_Request_type — The type of the PHDC request (**SET_FEATURE / CLEAR_FEATURE / GET_STATUS**). This parameter is only used by the usb_class_phdc_send_control_request function.

metadata — Boolean indicating a metadata send transfer. This parameter is only used by the usb_class_phdc_send_data function.

QoS — The qos to receive transfers. Used only by the usb_class_phdc_recv_data function.

buffer_ptr — Pointer to the buffer used in the transfer. This parameter is only used by the send and receive functions (**usb_class_phdc_send_data / usb_class_phdc_recv_data**).

buff_size — The size of the buffer used for transfer. This parameter is only used by the send and receive functions (**usb_class_phdc_send_data / usb_class_phdc_recv_data**).

tr_index — Unique index which identifies the transfer after it is queued in the USB host API lower layers. This parameter is written by the PHDC in case of a Send / Receive transfer (only if **USB_STATUS** is **USB_OK**).

tr_pipe_handle — The handle on which the transfer was queued. This parameter is written by PHDC in case of a Send / Receive transfer (only if **USB_STATUS** is **USB_OK**).

usb_status — Standard **USB_STATUS** when the transfer is finished (the application callback is called). This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. It is not valid until the corresponding callback is called.

usb_phdc_status — The PHDC specific status code for the current transaction. This parameter can take the following values: PHDC specific status codes. This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. It is not valid until the corresponding callback is called.

Chapter 7 Data Types

7.1 Data Types

The following table lists the USB Host API data types.

Table 7-1. USB Host API data types

USB Device API data type	Simple data type
_usb_host_handle	void*
_pipe_handle **_usb_device_instance_handle** **_usb_interface_descriptor_handle**	void* void* void*