Freescale MQX™ RTOS USB Device API Reference Manual

MQXUSBDEVAPI
Rev. 3
02/2014

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXUSBDEVAPI
Rev. 3, 02/2014

freescale™

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. To verify you have the latest information available, refer to freescale.com and navigate to Design Resources>Software and Tools>All Software 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. 0	01/2009	Initial Release coming with MQX RTOS version 3.0
Rev. 1	12/2011	“USB Device Layer API”, “USB Device Class API”, “USB Descriptor API” and “Data Structures” sections added.
Rev. 2	06/2013	Grammatical and stylistic corrections.
Rev. 3	10/2013	Updated content to reflect the switch from MQX types to C99 types.

Chapter 1
Before You Begin

Chapter 2
Overview

Chapter 3
USB Device Layer API

Chapter 4
USB Device Class API

Chapter 5
USB Descriptor API

Chapter 6
Data Structures

Chapter 7
Reference Data Types

Chapter 1
Before You Begin

1.1 About This Book

This document describes the USB Device driver and the programming interface as it is implemented in the MQX™ RTOS.

The authors assume that the user is familiar with the following reference material:

- *Universal Serial Bus Specification Revision 1.1*
- *Universal Serial Bus Specification Revision 2.0*

Use this book in addition to:

- *Freescale MQX™ RTOS User's Guide*
- *Freescale MQX™ RTOS API Reference Manual*
- *Freescale MQX™ RTOS USB Host User's Guide*
- *Source Code*

1.2 About MQX RTOS

MQX RTOS is real-time operating system from MQX Embedded. It is 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. Compared to the original MQX distributions, Freescale MQX distribution is simpler to configure and use. One release now contains MQX operating system in addition to other software components supported for a given microprocessor part (such as network or USB communication stacks). The first Freescale MQX RTOS release is assigned a number 3.0. It is based on, and is API-level compatible with, the MQX RTOS 2.50 released by ARC.

In this document, MQX RTOS is used as an abbreviation for the 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
HID	Human Interface Device
MSD	Mass Storage Device

Table 1-1. Acronyms and abbreviations (continued)

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 Function Listing Format

This is the general format of an entry for a function, compiler intrinsic, or a 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] — Pointer to x

parameter_2 [out] — Handle for y

parameter_n [in/out] — Pointer to z

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.

Description — Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- function blocks or might block under certain conditions
- function must be started as a task
- function creates a task
- function has pre-conditions that might not be obvious
- function has restrictions or special behavior

Return value — Specifies any value or values returned by function **function_name()**.

See also — Lists other functions or data types related to function **function_name()**.

Example — Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

Chapter 2
Overview

2.1 USB at a Glance

USB (Universal Serial Bus) is a polled bus. USB Host configures devices attached to it, either directly or through a USB hub, and initiates all bus transactions. USB Device responds only to the requests sent to it by a USB Host.

USB Device software consists of the:

- USB Device application
- USB Device Driver (contains USB Device Class APIs)
- USB Device APIs (independent of hardware)
- USB Device controller interface (DCI) - low-level functions used to interact with the USB Device controller hardware

2.2 Interaction Between USB Host and USB Device

Freescale MQX USB Device API includes the following components:

- USB Device APIs
- USB Device controller interface (DCI)
- An example of a USB specification’s Chapter 9 (device framework) responder
- USB Class APIs

Figure 2-1 shows the interaction between a USB Host and a USB Device.

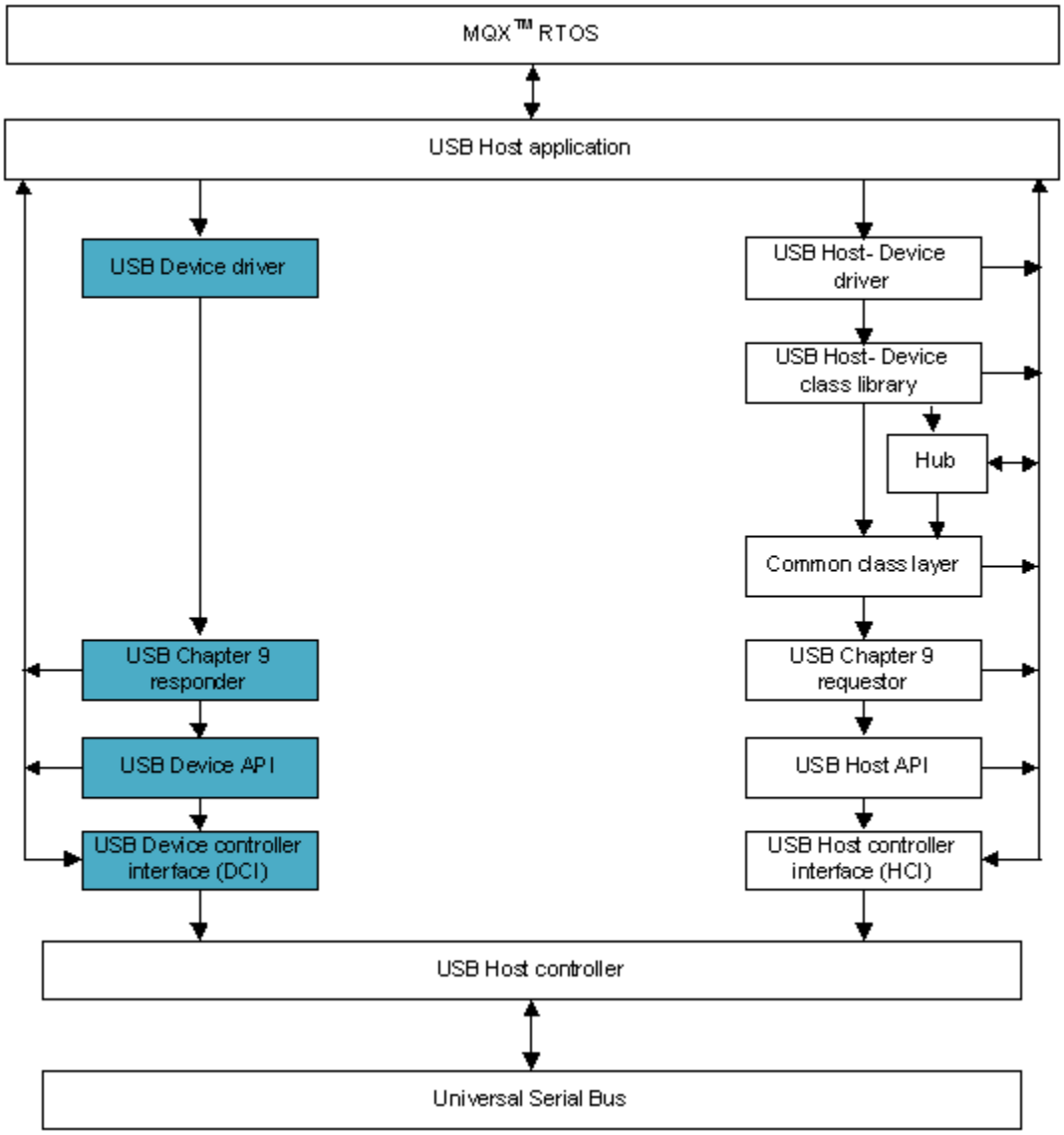

Figure 2-1. USB Host and USB Device Interaction

2.3 API Overview

This section describes the list of API functions and their use.

Table 2-1 summarizes the USB Device APIs.

Table 2-1. Summary of USB Device APIs

No.	API function	Description
1	USB_Device_Assert_Resume()	Resumes signal on the bus for remote wake-up
2	USB_Device_Cancel_Transfer()	Cancels a pending send or receive call

Table 2-1. Summary of USB Device APIs (continued)

No.	API function	Description
3	USB_Device_DeInit_EndPoint()	Disables the previously initialized endpoint passed as parameter
4	USB_Device_Get_Status()	Gets the internal USB device state
5	USB_Device_Get_Transfer_Status()	Gets the status of the last transfer on a particular endpoint
6	USB_Device_Init()	Initializes a USB device controller
7	USB_Device_Init_EndPoint()	Initializes the endpoint provided as parameter to the API
8	USB_Device_Read_Setup_Data()	Reads the setup data for an endpoint
9	USB_Device_Recv_Data()	Copies the data received on an endpoint and sets the endpoint to receive the next set of data
10	USB_Device_Register_Service()	Registers the callback service for a type of event or endpoint
11	USB_Device_Send_Data()	Sends data on an endpoint
12	USB_Device_Set_Address()	Sets the address of a USB device controller
13	USB_Device_Set_Status()	Sets the internal USB device state
14	USB_Device_Shutdown()	Shuts down a USB device controller
15	USB_Device_Stall_EndPoint()	Stalls an endpoint in the specified direction
16	USB_Device_Unstall_EndPoint()	Un-stalls a previously stalled endpoint
17	USB_Device_Unregister_Service()	Un-registers the callback service for a type of event or endpoint

Table 2-2 summarizes the common class APIs

Table 2-2. Summary of common class APIs

No.	API function	Description
1	USB_Class_Init()	The function initializes the Class Module
2	USB_Class_Send_Data()	The function calls the device to send data upon receiving an IN token
3	USB_Class_Get_Desc()	This function is called in to get the descriptor as specified in command
4	USB_Class_Set_Desc()	This function is called in to Set the descriptor as specified in command

Table 2-3 summarizes the CDC class APIs.

Table 2-3. Summary of CDC class APIs

No.	API function	Description
1	USB_Class_CDC_Init()	Initializes the CDC class
2	USB_Class_CDC_Recv_Data()	Receives the data from the host
3	USB_Class_CDC_Send_Data()	Send the data to the host
4	USB_Class_CDC_Periodic_Task()	Periodic call to the class driver to complete pending tasks

Table 2-4 summarizes the HID class APIs.

Table 2-4. Summary of HID class APIs

No.	API function	Description
1	USB_Class_HID_Init()	Initializes the HID class
2	USB_Class_HID_Send_Data()	Sends the HID report to the host
3	USB_Class_HID_Periodic_Task()	Periodic call to the class driver to complete pending tasks

Table 2-5 summarizes the MSC class APIs.

Table 2-5. Summary of MSC class APIs

No.	API function	Description
1	USB_Class_MSC_Init()	Initializes the MSC class
2	USB_Class_MSC_Periodic_Task()	Periodic call to the class driver to complete pending tasks

Table 2-6 summarizes the PHDC class APIs.

Table 2-6. Summary of PHDC class APIs

No.	API function	Description
1	USB_Class_PHDC_Init()	Initializes the PHDC class
2	USB_Class_PHDC_Send_Data()	Sends the PHDC report to the host
3	USB_Class_PHDC_Recv_Data()	Receives data from the PHDC Receive Endpoint of desired QOS
4	USB_Class_PHDC_Periodic_Task()	Periodic call to the class driver to complete pending tasks

Table 2-7 summarizes the descriptor module API functions required by the class layers for application implementation. See Chapter 5, “USB Descriptor API for more details about sample implementation of each API function.

Table 2-7. Summary of Descriptor Module API functions

No.	API function	Description;
1	USB_Desc_Get_Descriptor()	Gets various descriptors from the application
2	USB_Desc_Get_Endpoints()	Gets the endpoints used and their properties
3	USB_Desc_Get_Interface()	Gets the currently configured interface
4	USB_Desc_Remote_Wakeup()	Checks whether the application supports remote wake-up or not
5	USB_Desc_Set_Interface()	Sets new interface
6	USB_Desc_Valid_Configation()	Checks whether the configuration being set is valid or not
7	USB_Desc_Valid_Interface()	Checks whether the interface being set is valid or not

2.4 Using the USB Device API

This section describes how to use various device and class API functions.

2.4.1 Using the Device Layer API

This section describes how to use the device layer API functions from the class driver or the monolithic application.

2.4.1.1 Initialization flow

To initialize the driver layer, the class driver must:

1. Call _usb_device_init() to initialize the low level driver and the controller.
2. Call _usb_device_register_service() to register service callback functions for the following bus event:
   - USB_SERVICE_BUS_RESET
   - USB_SERVICE_SUSPEND
   - USB_SERVICE_SOF
   - USB_SERVICE_RESUME
   - USB_SERVICE_SLEEP
   - USB_SERVICE_ERROR
   - USB_SERVICE_STALL
3. Call _usb_device_register_service() to register service call back functions for control and non-control endpoints (endpoint events).
4. Call _usb_device_init_endpoint() to initialize the control endpoint and endpoints used by the application.
5. The device layer must be initialized to send callbacks registered in any event on the USB bus. The devices must start receiving the USB Chapter 9 framework calls on control endpoint. The lower layer driver propagates these calls to the class driver.

2.4.1.2 Transmission flow

After the initialization, the class driver can call the low level send routine to transmit data. The transmission process includes the following steps:

1. The class driver calls _usb_device_send_data() to start the transmission by passing the endpoint number, size, and buffer to the call.
2. As soon as the controller completes the transfer, a call is made to the service callback registered to the particular endpoint.

2.4.1.3 Receive flow

After the initialization, the class driver must be ready to receive data. The receive process includes the following steps:

1. When the data is received at the configured endpoint, the low level driver calls the service registered using _usb_device_register_service() to that endpoint passing it the buffer and size of the data received.

2. The class driver calculates the size of the complete packet from the data in the buffer and makes a call to the _usb_device_recv_data() to receive the complete packet. To do so, it passes the class driver buffer pointer and complete packet size to receive the data. When the complete packet size is equal to the data received, it processes the packet. Otherwise, it waits to receive the complete packet in the next callback to process it.

2.4.2 CDC Class Layer API

To use CDC class layer API functions from the application:

1. Call USB_Class_CDC_Init() to initialize the class driver, all the layers below it, and the device controller. Event callback functions are also passed as parameter to this function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application should move into the connected state.
3. Call USB_Class_CDC_Send_Data() to send data to the host through the device layers, when required.
4. Call USB_Class_CDC_Recv_Data() when callback function is called with the USB_APP_DATA_RECEIVED event (that implies reception of data from the host).

2.4.3 HID Class Layer API

To use HID class layer API functions from the application:

1. Call USB_Class_HID_Init() to initialize the class driver, all the layers below it, and the device controller. Event callback functions are also passed as a parameter to this function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application should move into the ready state.
3. Call USB_Class_HID_Send_Data() to send data to the host through the device layers, when required.

2.4.4 MSC Class Layer API

To use MSD class layer API functions from the application:

1. Call USB_Class_MSC_Init() to initialize the class driver, all the layers below it, and the device controller. Event callback functions are also passed as a parameter to this function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application should move into the ready state.
3. Callback function is called with the USB_MSC_DEVICE_READ_REQUEST event to copy data from the storage device before sending it to the USB bus. It reads data from the mass storage device to the driver buffer.
4. Callback function is called with the USB_MSC_DEVICE_WRITE_REQUEST event to copy data from the USB driver buffer to the Storage device. It reads data from the driver buffer to the mass storage device.

2.4.5 PHDC Class Layer API

To use PHDC class layer API functions from the application:

1. Call USB_Class_PHDC_Init() to initialize the class driver, all the layers below it, and the device controller. Event callback functions are also passed as parameter to this function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application should move into the connected state.
3. Call USB_Class_PHDC_Send_Data() to send data to the host through the device layers, when required.
4. Call USB_Class_PHDC_Recv_Data() when callback function is called with the USB_APP_DATA_RECEIVED event (that implies reception of data from the host).

Chapter 3
USB Device Layer API

3.1 USB Device Layer API function listings

3.1.1 _usb_device_assert_resume()

Resume the USB Host.

Synopsis

```
void  _usb_device_assert_resume
(
        _usb_device_handle        handle
);
```

Parameters

handle [in] — USB Device handle

Description

The function sends a resume signal on the USB bus for remote wakeup. This function is called when the device needs to send data to the USB host and the USB bus is in suspend state. Blocks for 20 ms until the resume assertion is complete.

Return value

None

See also:

_usb_device_init()

_usb_device_init_endpoint()

3.1.2 _usb_device_cancel_transfer()

Cancel the transfer on the endpoint.

Synopsis

```
uint8_t  _usb_device_cancel_transfer
(
        _usb_device_handle        handle,
        uint8_t                   endpoint_number,
        uint8_t                   direction
);
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number for the transfer

direction [in] - Direction of transfer; one of:

USB_RECV

USB_SEND

Description

The function checks whether the transfer on the specified endpoint and direction is active. If it is not active, the function changes the status to idle and returns. If the transfer is active, the function calls the DCI function to terminate all transfers queued on the endpoint and sets the status to idle.

This function blocks until the transfer cancellation at the hardware is completed.

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

See Also:

_usb_device_get_transfer_status()

_usb_device_init()

_usb_device_init_endpoint()

3.1.3 _usb_device_deinit_endpoint()

Disable the endpoint for the USB Device controller.

Synopsis

```
uint8_t  _usb_device_deinit_endpoint
(
        _usb_device_handle      handle,
        uint8_t                 endpoint_number,
        uint8_t                 direction
)
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number

direction [in] - Direction of transfer; one of:

USB_RECV

USB_SEND

Description

The function resets the data structures specific to the specified endpoint and calls the DCI function to disable the endpoint in the specified direction.

Return value

- **USB_OK** (success)
- **USBERR_ERROR** (failure: endpoint deinitialization failed)

See Also:

_usb_device_init_endpoint()

3.1.4 _usb_device_get_status()

Get the internal USB device state.

Synopsis

```
uint8_t  _usb_device_get_status
(
        _usb_device_handle  handle,
        uint8_t              component,
        uint16_t * status
)
```

Parameters

handle [in] - USB Device handle

component [in] - Component status to get; one of:

USB_STATUS_ADDRESS

USB_STATUS_CURRENT_CONFIG

USB_STATUS_DEVICE

USB_STATUS_DEVICE_STATE

USB_STATUS_ENDPOINT - The LSB nibble carries the endpoint number

USB_STATUS_INTERFACE

USB_STATUS_SOF_COUNT

status [out] - Requested status

Description

The function gets the status of the specified component for the GET STATUS device request. This function must be used by the GET STATUS device response function.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: incorrect component status requested)
- **USBERR_ERROR** (failure: unknown error)

See Also:

_usb_device_set_status()

3.1.5 _usb_device_get_transfer_status()

Get the status of the last transfer on the endpoint.

Synopsis

```
uint8_t  _usb_device_get_transfer_status
(
    _usb_device_handle  handle,
    uint8_t             endpoint_number,
    uint8_t             direction
)
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number

direction [in] - Direction of transfer; one of:

USB_RECV

USB_SEND

Description

The function gets the status of the transfer on the endpoint specified by *endpoint_number*. It reads the status and also checks whether the transfer is active. If the transfer is active, depending on the hardware, the function may call the DCI function to check the status of that transfer.

To check whether a receive or send transfer was complete, the application can call **_usb_device_get_transfer_status()** or use the callback function registered for the endpoint.

Return Value

- Status of the transfer; one of:

  USB_STATUS_TRANSFER_IN_PROGRESS (transfer is active on the specified endpoint)

  USB_STATUS_DISABLED (endpoint is disabled)

  USB_STATUS_IDLE (endpoint is idle)

  USB_STATUS_STALLED (endpoint is stalled)

  USBERR_ERROR (failure: unknown error)

See Also:

_usb_device_init()

_usb_device_init_endpoint()

_usb_device_recv_data()

_usb_device_send_data()

3.1.6 _usb_device_init()

Initialize the USB Device controller.

Synopsis

```
uint8_t  _usb_device_init
(
    uint8_t                     device_number,
    _usb_device_handle  *  handle,
    uint8_t                     number_of_endpoints
);
```

Parameters

device_number [in] - USB Device controller to initialize

handle [out] - Pointer to a USB Device handle

number_of_endpoints [in] - Number of endpoints to initialize

Description

The function does the following:

- Initializes the USB Device-specific data structures
- Initializes the status for all transfer data structures to **USB_STATUS_DISABLED**
- Changes the device state from **USB_UNKNOWN_STATE** to **USB_POWERED_STATE**
- Calls the device-specific initialization function
- Installs the interrupt service routine for USB interrupts

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_DEVICE_NUM** (failure: invalid USB device controller)
- **USBERR_ALLOC_STATE** (failure: cannot allocate memory for USB device state structure)
- **USBERR_DRIVER_NOT_INSTALLED** (failure: USB callback structure is not initialized)
- **USBERR_UNKNOWN_ERROR** (failure: unknown error)
- **USBERR_ALLOC_TR** (failure: cannot allocate memory for endpoints' structure)
- **USBERR_ALLOC** (failure: cannot allocate memory for internal scratch structure)
- **USBERR_ERROR** (failure: USB device callback function pointer of DCI Device Init function is not initialized)
- **USBERR_INSTALL_ISR** (failure: cannot install USB interrupt)

See Also:

_usb_device_shutdown()

3.1.7 _usb_device_init_endpoint()

Initialize the endpoint for the USB Device controller.

Synopsis

```
uint8_t  _usb_device_init_endpoint
(
    _usb_device_handle      handle,
    USB_EP_STRUCT_PTR       ep_ptr,
    uint8_t                 flag
);
```

Parameters

handle [in] - USB Device handle

ep_ptr [in] - Pointer to the USB endpoint

flag [in] - One of:

0- if the last data packet transferred is MAX_PACKET_SIZE bytes, terminate the transfer with a zero-length packet

1 or 2 - maximum number of transactions per microframe (relevant only for USB 2.0 and high-bandwidth endpoints)

Description

The function initializes endpoint-specific data structures and calls the DCI function to initialize the specified endpoint.

Return Value

- **USB_OK** (success)
- **USBERR_EP_INIT_FAILED** - USB 2.0 Device API only (failure: endpoint initialization failed)
- **USBERR_ERROR** (failure: USB device callback function pointer of DCI Init Endpoint function is not initialized)
- **USBERR_ALLOC** (failure: cannot allocate memory)

See Also:

_usb_device_deinit_endpoint()

_usb_device_init()

3.1.8 _usb_device_read_setup_data()

Read the setup data for the endpoint.

Synopsis

```
uint8_t _usb_device_read_setup_data
(
    _usb_device_handle  handle,
    uint8_t             endpoint_number,
    unsigned char *           buffer_ptr
);
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number for the transaction

buffer_ptr [in/out] - Pointer to the buffer into which to read data

Description

Call the function only after the callback function for the endpoint notifies the application that a setup packet has been received. The function reads the setup packet, which USB Device API received by calling **_usb_device_recv_data()** internally.

Depending on the hardware, the function may call the DCI function to read the setup data from the endpoint.

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

See Also:

_usb_device_init()

_usb_device_init_endpoint()

_usb_device_recv_data()

3.1.9 _usb_device_recv_data()

Receive data from the endpoint.

Synopsis

```
uint8_t  _usb_device_recv_data
(
    _usb_device_handle  handle,
    uint8_t             endpoint_number,
    unsigned char *           buffer_ptr,
    uint32_t            size
)
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number for the transaction

buffer_ptr [in] - Pointer to the buffer into which to receive data

size [in] - Number of bytes to receive

Description

The function enqueues the receive request and returns.

To check whether the transaction was complete, the application can call **_usb_device_get_transfer_status()** or use the callback function registered for the endpoint.

Do not call **_usb_device_recv_data()** to receive a setup packet.

Return Value

- **USB_OK** (success)
- **USBERR_RX_FAILED** (failure: data reception from the endpoint failed)
- **USBERR_TRANSFER_IN_PROGRESS** (failure: Endpoint is stalled; no transfer can take place until the endpoint is unstalled)
- **USBERR_ERROR** (failure: other errors)

See Also:

_usb_device_get_transfer_status()

_usb_device_init()

_usb_device_init_endpoint()

3.1.10 _usb_device_register_service()

Register the service for the type of event or endpoint.

Synopsis

```
uint8_t  _usb_device_register_service
(
    _usb_device_handle  handle,
    uint8_t             event_endpoint,
    void (_CODE_PTR service)(USB_EVENT_STRUCT,void *),
    void * arg
);
```

Parameters

handle [in] - USB Device handle

event_endpoint [in] - Endpoint (0 through 15) or event to service. Event; one of:

USB_SERVICE_BUS_RESET

USB_SERVICE_ERROR

USB_SERVICE_RESUME

USB_SERVICE_SLEEP

USB_SERVICE_STALL

service [in] - Callback function that services the event or endpoint

Return Value

- **USB_OK** (success)
- **USBERR_ALLOC** (failure: could not allocate internal data structures for registering services)
- **USBERR_OPEN_SERVICE** (failure: service was already registered)

See Also:

_usb_device_unregister_service()

3.1.11 _usb_device_send_data()

Send data on the endpoint.

Synopsis

```
uint8_t  _usb_device_send_data
(
        _usb_device_handle  handle,
        uint8_t             endpoint_number,
        unsigned char *           buffer_ptr,
        uint32_t            size
)
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number of the transaction

buffer_ptr [in] - Pointer to the buffer to send

size [in] - Number of bytes to send

Description

The function calls the DCI function to send the data on the endpoint specified by *endpoint_number*. The function queues the sent request by passing the data size as a parameter along with the buffer pointer. When the complete data has been sent, the device layer sends an event to the calling function. This can be done only if a service for this endpoint has been registered. The buffer pointed to by the buffer pointer must not be used until the complete send data event is received.To check whether the transaction was complete, the application can call **_usb_device_get_transfer_status()** or use the callback function registered for the endpoint.

Return Value

- **USB_OK** (success)
- **USBERR_TRANSFER_IN_PROGRESS** (failure: previously queued transfer on the specified endpoint is still in progress; wait until the transfer has been completed; call **_usb_device_get_transfer_status()** to determine when the endpoint has a status of USB_STATUS_IDLE). Relevant to USB 1.1 stack only).
- **USBERR_TX_FAILED** (failure: data transfer from the endpoint failed)
- **USBERR_ERROR** (failure: other error)

See Also:

_usb_device_recv_data()

_usb_device_get_transfer_status()

3.1.12 _usb_device_set_address()

Set the address of the USB Device. Available in USB 2.0 Device API only.

Synopsis

```
uint8_t _usb_device_set_address
(
    _usb_device_handle  handle,
    uint8_t             address
);
```

Parameter

handle [in] - USB Device handle

address [in] - Address of the USB device

Description

The function calls the DCI function to initialize the device address. It can be called by set-address response functions. This API function is called only when the control transfer that carries the address as part of the setup packet from the host to the device has completed.

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

3.1.13 _usb_device_set_status()

Set the internal USB device state.

Synopsis

```
uint8_t  _usb_device_set_status
(
    _usb_device_handle  handle,
    uint8_t             component,
    uint16_t            setting
);
```

Parameters

handle [in] - USB Device handle

component [in] - Component status to set (see **_usb_device_get_status()**)

status [in] - Status to set

Description

The function sets the status of the specified component for the SET STATUS device request. This function must be used by the SET STATUS device response function.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: incorrect component status requested)
- **USBERR_ERROR** (failure: other errors)

See Also:

_usb_device_get_status()

3.1.14 _usb_device_shutdown()

Shuts down the USB Device controller.

Synopsis

```
uint8_t _usb_device_shutdown
(
    _usb_device_handle   handle
);
```

Parameters

handle [in] - USB Device handle

Description

The function is useful if the services of the USB Device controller are no longer required or if the USB Device controller needs to be configured as a host.

The function does the following:

1. Terminates all transactions
2. Un-registers all the services
3. Disconnects the device from the USB bus

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

See Also:

_usb_device_init()

3.1.15 _usb_device_stall_endpoint()

Stall the endpoint in the specified direction.

Synopsis

```
uint8_t _usb_device_stall_endpoint
(
    _usb_device_handle  handle,
    uint8_t             endpoint_number,
    uint8_t             direction
);
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number to stall

direction [in] - Direction to stall; one of:

USB_RECV

USB_SEND

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

See Also:

_usb_device_unstall_endpoint()

3.1.16 _usb_device_unregister_service()

Un-register the service for the type of event or endpoint.

Synopsis

```
uint8_t _usb_device_unregister_service
(
    _usb_device_handle  handle,
    uint8_t             event_endpoint
);
```

Parameters

handle [in] - USB Device handle

event_endpoint [in] - Endpoint (0 through 15) or event to service (see **_usb_device_register_service()**)

Description

The function un-registers the callback function that is used to process the event or endpoint. As a result, that type of event or endpoint cannot be serviced by a callback function.

Before calling the function, the application must disable the endpoint by calling **_usb_device_deinit_endpoint()**.

Return Value

- **USB_OK** (success)
- **USBERR_CLOSED_SERVICE** (failure: service was not previously registered)
- **USBERR_ERROR** (failure: other errors)

See Also:

_usb_device_deinit_endpoint()

_usb_device_register_service()

3.1.17 _usb_device_unstall_endpoint()

Unstall the endpoint in the specified direction.

Synopsis

```
uint8_t_usb_device_unstall_endpoint
(
        _usb_device_handle  handle,
        uint8_t              endpoint_number,
        uint8_t              direction
);
```

Parameters

handle [in] - USB Device handle

endpoint_number [in] - Endpoint number to unstall

direction [in] - Direction to unstall; one of:

USB_RECV

USB_SEND

Return Value

- **USB_OK** (success)
- **USBERR_ERROR** (failure)

See Also:

_usb_device_stall_endpoint()

Chapter 4
USB Device Class API

This section discusses the API functions provided as part of class implementations.

4.1 Common Class API function listings

4.1.1 USB_Class_Init()

Initialize the class module.

Synopsis

```
USB_CLASS_HANDLE USB_Class_Init
(
    _usb_device_handle        handle,
    USB_CLASS_CALLBACK        class_callback,
    USB_REQ_FUNC              other_req_callback,
    void *                    user_arg,
    DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr
);
```

Parameters

handle [in] - USB device controller to initialize

class_callback [in] - class callback function pointer

other_req_callback[in] - vendor specific callback function pointer

user_arg[in] - parameter to be passed to class callback function

desc_callback_ptr[in] - pointer to a structure of the descriptor function pointers

Description

The function initializes the class state object and registers service for USB events.

Return Value

- **class handle** (success)
- **others** (failure)

4.1.2 USB_Class_Send_Data()

Sends data to the host.

Synopsis

```
uint8_t USB_Class_Send_Data
(
    USB_CLASS_HANDLE    handle,
    uint8_t             ep_num,
    uint8_t *           buff_ptr,
    uint32_t            size
)
```

Parameters

handle [in] - class handle returned by **USB_Class_Init()**

ep_num [in] - endpoint number

buff_ptr [in] - buffer to send

size [in] - length of the transfer

Description

This function is called to send data upon receiving an IN token.

Return Value

- **USB_OK** (success)
- **others** (failure)

4.1.3 USB_Class_Get_Desc()

Get the descriptor.

Synopsis

```
uint8_t USB_Class_Get_Desc
(
    USB_CLASS_HANDLE  handle,
    int32_t           cmd,
    uint8_t           input_data,
    uint8_t *         *out_buf
}
```

Parameters

handle [in] - class handle returned by **USB_Class_Init()**

cmd [in] - command for USB descriptor to get

input_data [in] - input to the application function

out_buf [out] - buffer to get descriptoror to

Description

The function returns device descriptor. This functions is called when a GET request is received from the host.

Return Value

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

4.1.4 USB_Class_Set_Desc()

Set the descriptor.

Synopsis

```
uint8_t USB_Class_Get_Desc
(
    USB_CLASS_HANDLE  handle,
    int32_t           cmd,
    uint8_t           input_data,
    uint8_t *         *in_buf
}
```

Parameters

handle [in] - class handle returned by the **USB_Class_Init()**

cmd [in] - command for the USB descriptor to set

input_data [in] - input to the application function

in_buf [in] - buffer containing a descriptor to set

Description

This functions is called when a SET request is received from host.

Return Value

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

4.2 CDC Class API function listings

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

4.2.1 USB_Class_CDC_Init()

Initialize the CDC class.

Synopsis

```
uint8_t USB_Class_CDC_Init
(
    CDC_CONFIG_STRUCT_PTR cdc_config_ptr
);
```

Parameters

cdc_config_ptr [in] - pointer to the configuration parameter sent by the API to configure the CDC class

Description

Application calls this API function to initialize the CDC class, the underlying layers, and the controller hardware.

Return Value

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

See Also:

CDC_CONFIG_STRUCT

4.2.2 USB_Class_CDC_Send_Data()

Send CDC data.

Synopsis

```
uint8_t USB_Class_CDC_Send_Data
(
    CDC_HANDLE     handle,
    uint8_t        ep_num,
    uint8_t *      app_buff,
    uint32_t       size
);
```

Parameters

handle [in] - handle returned by **USB_Class_CDC_Init()**

ep_num[in] - endpoint number

app_buff[in] - buffer to send

size[in] - length of the transfer

Description

The application calls this API function to send DIC data specified by *app_buff* and *size*. Data is sent through DIC_SEND_ENDPOINT. Once the data has been sent, the application layer receives a callback event. The application reserves the buffer until it receives a callback event stating that the data has been sent.

Return Value

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

See Also:

USB_Class_CDC_Init()

4.2.3 USB_Class_CDC_Recv_Data()

Receive CDC data.

Synopsis

```
uint8_t USB_Class_CDC_Recv_Data
(
    CDC_HANDLE        handle,
    uint8_t           ep_num,
    uint8_t *         buff_ptr,
    uint32_t          size
);
```

Parameters

handle [in] - handle returned by **USB_Class_CDC_Init()**

ep_num[in] - endpoint number

buff_ptr[out] - buffer to receive

size[in] - Number of bytes to receive

Description

The function calls this API function to receive CDC report data in the specified *buff_ptr* of length given by *size*. Data is received through DIC_RECV_ENDPOINT. Once the data has been received, the application layer receives a callback event. The application reserves the buffer until it receives a callback event stating that the data has been received.

Return Value

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

See Also:

USB_Class_CDC_Init()

4.2.4 USB_CDC_Periodic_Task()

Complete any left over activity during a specified time period.

Synopsis

```
void USB_Class_CDC_Periodic_Task(void);
```

Parameters

None

Description

The application calls this API function to enable the driver to complete any left over activity on the device's control endpoint.

Return Value

None

4.3 HID Class API function listings

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

4.3.1 USB_Class_HID_Init()

Initialize the HID class.

Synopsis

```
uint8_t USB_Class_HID_Init
(
    HID_CONFIG_STRUCT_PTR hid_config_ptr
);
```

Parameters

hid_config_ptr [in] - pointer to the configuration parameter sent by the API to configure the HID class

Description

The application calls this API function to initialize the HID class, the underlying layers, and the controller hardware.

Return Value

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

See Also:

HID_CONFIG_STRUCT

4.3.2 USB_Class_HID_Send_Data()

Send HID data.

Synopsis

```
uint8_t USB_Class_HID_Send_Data
(
    HID_HANDLE        handle,
    uint8_t           ep_num,
    uint8_t *         app_buff,
    uint32_t          size
);
```

Parameters

handle [in] - handle returned by **USB_Class_HID_Init()**

ep_num[in] - endpoint number

app_buff[in] - buffer to send

size[in] - length of the transfer

Description

The function calls this API to send HID report data specified by *app_buff* and *size*. Once the data has been sent, the application layer receives a callback event. The application reserves the buffer until it receives a callback event stating that the data has been sent.

Return Value

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

See Also:

USB_Class_HID_Init()

4.3.3 USB_HID_Periodic_Task()

Complete any left over activity during a specified time period.

Synopsis

```
void USB_Class_HID_Periodic_Task(void);
```

Parameters

None

Description

The application calls this API function to enable the class driver to complete any left over activity on the device's control endpoint.

Return Value

None

4.4 MSC Class API function listings

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

4.4.1 USB_Class_MSC_Init()

Initialize the MSC class.

Synopsis

```
uint8_t USB_Class_MSC_Init
(
    USB_MSD_CONFIG_STRUCT_PTR msd_config_ptr
)
```

Parameters

usb_msd_config_ptr [in] - pointer to the configuration parameter send by the API to configure the MSC class

Description

The application calls this API function to initialize the MSC class, the underlying layers, and the controller hardware.

Return Value

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

See Also:

USB_MSD_CONFIG_STRUCT

4.4.2 USB_MSC_Periodic_Task()

Complete any left over activity during a specified time period.

Synopsis

```
void USB_Class_MSC_Periodic_Task(void);
```

Parameters

None

Description

The application calls this API function to enable the class driver to complete any left over activity on the device's control endpoint.

Return Value

None

4.5 PHDC Class API function listings

This section defines API functions used for the Personal Healthcare Device Class (PHDC). The user can employ these API functions to make PHDC applications.

4.5.1 USB_Class_PHDC_Init()

Initialize the PHDC class.

Synopsis

```
uint8_t USB_Class_PHDC_Init
(
    PHDC_CONFIG_STRUCT_PTR phdc_config_ptr
);
```

Parameters

phdc_config_ptr [in] - pointer to the configuration parameter sent by the API to configure the PHDC class

Description

The application calls this API function to initialize the PHDC class, the underlying layers, and the controller hardware.

Return Value

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

See Also:

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

DESC_CALLBACK_FUNCTIONS_STRUCT

USB_ENDPOINTS

4.5.2 USB_Class_PHDC_Send_Data()

Sends the PHDC report to the host.

Synopsis

```
uint8_t USB_Class_PHDC_Send_Data
(
    PHDC_HANDLE handle,
    bool    meta_data,
    uint8_t num_tfr,
    uint8_t qos,
    uint8_t *app_buff,
    uint32_t size
);
```

Parameters

handle [in] - handle returned by **USB_Class_PHDC_Init()**

meta_data[in] - packet is meta data or not

num_tfr[in] - number of transfer

qos[in] - current qos of the transfer

app_buff[in] - buffer to send

size[in] - length of the transfer

Description

The function calls this API function to send PHDC report data specified by *meta_data*, *num_tfr*, *qos*, *app_buff*, and *size*. Once the data has been sent, the application layer receives a callback event. The application reserves the buffer until it receives a callback event stating that the data has been sent.

Return Value

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

See Also:

USB_Class_PHDC_Init()

4.5.3 USB_Class_PHDC_Recv_Data()

Receives data from the PHDC receive endpoint of desired QOS.

Synopsis

```
uint8_t USB_Class_PHDC_Recv_Data
(
    PHDC_HANDLE handle,
    uint8_t qos,
    uint8_t * buff_ptr,
    uint32_t size
);
```

Parameters

handle [in] - handle returned by **USB_Class_PHDC_Init()**

qos[in] - QOS of the transfer

buff_ptr[out] - buffer to receive

size[in] - number of bytes to receive

Description

The function is used to receive PHDC data from the endpoint specified by current_qos. This function uses **_usb_device_recv_data()** function to perform the required functionality.

Return Value

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

See Also:

_usb_device_recv_data()

USB_Class_PHDC_Init()

4.5.4 USB_PHDC_Periodic_Task()

Complete any left over activity during a specified time period.

Synopsis

```
void USB_Class_PHDC_Periodic_Task(void);
```

Parameters

None

Description

The application calls this API function to enable the class driver to complete any left over activity on the device's control endpoint.

Return Value

None

Chapter 5
USB Descriptor API

This section discusses API functions that are implemented as part of the application.

5.1 USB Descriptor API function listings

5.1.1 USB_Desc_Get_Descriptor()

Gets various descriptors from the application.

Synopsis

```
uint8_t USB_Desc_Get_Descriptor
(
    uint32_t               handle,
    uint8_t                type,
    uint8_t                str_num,
    uint8_t                index,
    uint16_t               *descriptor,
    uint8_t                *handle,
    USB_PACKET_SIZE        *size
);
```

Parameters

handler [in] - USB class handle

type [in] - type of descriptor requested

str_num [in] - string number for string descriptor

index [in] - string descriptor language ID

descriptor [out] - output descriptor pointer

size [out] - size of descriptor returned

Description

The framework module calls this function to get the descriptor information when Get_Descriptor framework call is received from the host.

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_REQ_TYPE** (failure: invalid request)

Sample Implementation:

```
uint8_t USB_Desc_Get_Descriptor(
         uint32_t handle,         /* [IN] handle */
         uint8_t type,            /* [IN] type of descriptor requested */
         uint8_t str_num,         /* [IN] string index for string descriptor */
         uint16_t index,          /* [IN] string descriptor language Id */
         uint8_t **descriptor, /* [OUT] output descriptor pointer */
```

```
     USB_PACKET_SIZE *size   /* [OUT] size of descriptor returned */
)
{
switch(type)
{
case USB_REPORT_DESCRIPTOR:
{
    type = USB_MAX_STD_DESCRIPTORS;
    *descriptor = (uint8_t *)g_std_descriptors [type];
    *size = g_std_desc_size[type];
}
break;
case USB_HID_DESCRIPTOR:
{
    type = USB_CONFIG_DESCRIPTOR ;
    *descriptor = (uint8_t *)(g_std_descriptors [type]+
                              CONFIG_ONLY_DESC_SIZE+IFACE_ONLY_DESC_SIZE);
    *size = HID_ONLY_DESC_SIZE;
}
break;
case USB_STRING_DESCRIPTOR:
{
    if(index == 0)
    {
        /* return the string and size of all languages */
        *descriptor = (uint8_t *)g_languages.languages_supported_string;
        *size = g_languages.languages_supported_size;
    } else
    {
        uint8_t lang_id=0;
        uint8_t lang_index=USB_MAX_LANGUAGES_SUPPORTED;

        for(;lang_id< USB_MAX_LANGUAGES_SUPPORTED;lang_id++)
        {
         /* check whether we have a string for this language */
         if(index == g_languages.usb_language[lang_id].language_id)
         {
             /* check for max descriptors */
             if(str_num < USB_MAX_STRING_DESCRIPTORS)
             {
                 /* setup index for the string to be returned */
                 lang_index=str_num;
             }
          break;
         }
        }
        /* set return val for descriptor and size */
        *descriptor = (uint8_t
*)g_languages.usb_language[lang_id].lang_desc[lang_index];
        *size =
g_languages.usb_language[lang_id].lang_desc_size[lang_index];
    }
}
```

```
        break;
        default :
            if (type < USB_MAX_STD_DESCRIPTORS)
            {
                 /* set return val for descriptor and size*/
                *descriptor = (uint8_t *)g_std_descriptors [type];
                /* if there is no descriptor then return error */
                if(*descriptor == NULL)
                {
                 return USBERR_INVALID_REQ_TYPE;
                }
                 *size = g_std_desc_size[type];
            }
            else /* invalid descriptor */
            {
                return USBERR_INVALID_REQ_TYPE;
            }
        break;
        }
        return USB_OK;
    }
```

5.1.2 USB_Desc_Get_Endpoints()

Gets the endpoints and their properties.

Synopsis

```
uint8_t USB_Desc_Get_Endpoints
(
    uint32_t handle
);
```

Parameters

handler [in] - USB class handle

Description

The class driver calls this function to get information about the non-control endpoints. The class driver can use this information to initialize these endpoints.

Return Value

Pointer to the structure containing information about the non-control endpoints.

Sample Implementation:

```
void* USB_Desc_Get_Endpoints(
      uint32_t handle /* [IN] handle */
)
{
    return (void*)&usb_desc_ep;
}
```

See also:

USB_ENDPOINTS

5.1.3 USB_Desc_Get_Interface()

Gets the currently configured interface.

Synopsis

```
uint8_t USB_Desc_Get_Interface
(
    uint32_t handle,
    uint8_t interface,
    uint8_t * alt_interface
);
```

Parameters

handler [in] - USB class handle

interface [in] - Interface number

alt_interface [out] - Output alternate interface

Description

The framework module calls this function to get the alternate interface corresponding to the interface provided as an input parameter.

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_REQ_TYPE** (failure: invalid request)

Sample Implementation:

```
uint8_t USB_Desc_Get_Interface
(
    uint32_t handle,            /* [IN] handle */
    uint8_t interface,          /* [IN] interface number */
    uint8_t * alt_interface     /* [OUT] output alternate interface */
)
{
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        /* get alternate interface*/
        *alt_interface = g_alternate_interface[interface];
        return USB_OK;
    }
    return USBERR_INVALID_REQ_TYPE;
}
```

5.1.4 USB_Desc_Remote_Wakeup()

Checks whether the application supports remote wake-up or not.

Synopsis

```
uint8_t USB_Desc_Remote_Wakeup
(
    uint32_t handle
);
```

Parameters

handler [in] - USB class handle

Description

This function is called by framework module. This function returns the boolean value as to whether the controller device supports remote wake-up or not.

Return Value

- **TRUE** (Remote wake-up supported)
- **FALSE** (Remote wake-up not supported)

Sample Implementation:

```
bool USB_Desc_Remote_Wakeup
(
```

```
    uint32_t handle /* [IN] handle */
)
{
    return REMOTE_WAKEUP_SUPPORT;
}
```

See also:

USB_ENDPOINTS

5.1.5 USB_Desc_Set_Interface()

Sets new interface.

Synopsis

```
uint8_t USB_Desc_Set_Interface
(
    uint32_t handle,
    uint8_t interface,
    uint8_t * alt_interface
);
```

Parameters

handler [in] - USB class handle

interface [in] - Interface number

alt_interface [in] - Input alternate interface

Description

The framework module calls this function to set the alternate interface corresponding to the interface provided as an input parameter. The alternate interface is also provided as an input parameter.

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_REQ_TYPE** (failure: invalid request)

Sample Implementation:

```
uint8_t USB_Desc_Set_Interface
(
    uint32_t handle,       /* [IN] handle */
    uint8_t interface,     /* [IN] interface number */
    uint8_t alt_interface  /* [IN] input alternate interface */
)
{
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        /* set alternate interface*/
        g_alternate_interface[interface]=alt_interface;
        return USB_OK;
    }
```

```
    return USBERR_INVALID_REQ_TYPE;
}
```

5.1.6 USB_Desc_Valid_Configation()

Checks if the configuration is valid.

Synopsis

```
uint8_t USB_Desc_Valid_Configation
(
    uint32_t handle,
    unit_16 config_val
);
```

Parameters

handler [in] - USB class handle

config_val [in] - USB descriptor configuration value

Description

This function is called by framework module to check whether the configuration is valid or not.

Return Value

- **TRUE** (Configuration is valid)
- **FALSE** (Configuration is invalid)

Sample Implementation:

```
bool USB_Desc_Valid_Configation
(
    uint32_t handle,      /*[IN] handle */
    uint16_t config_val   /*[IN] configuration value */
)
{
    uint8_t loop_index=0;
    /* check with only supported val right now */
    while(loop_index < (USB_MAX_CONFIG_SUPPORTED+1))
    {
        if(config_val == g_valid_config_values[loop_index])
        {
            return TRUE;
        }
        loop_index++;
    }
    return FALSE;
}
```

5.1.7 USB_Desc_Valid_Interface()

Checks if the interface is valid.

Synopsis

```
uint8_t USB_Desc_Valid_Interface
(
    uint32_t handle,
    uint8_t interface
);
```

Parameters

handler [in] - USB class handle

interface [in] - USB descriptor target interface

Description

This function is called by the class driver to check whether the interface is valid or not.

Return Value

- **TRUE** (Interface is valid)
- **FALSE** (Interface is invalid)

Sample Implementation:

```
bool USB_Desc_Valid_Interface
(
    uint32_t handle,    /*[IN] handle */
    uint8_t interface   /*[IN] target interface */
)
{
    uint8_t loop_index=0;
    /* check with only supported val right now */
    while(loop_index < USB_MAX_SUPPORTED_INTERFACES)
    {
        if(interface == g_alternate_interface[loop_index])
        {
            return TRUE;
        }
        loop_index++;
    }

    return FALSE;
}
```

Chapter 6
Data Structures

This section discusses the data structures that are passed as parameters in various API functions.

6.1 USB Device Layer Data Structure listings

6.1.1 _usb_device_handles

This data type is a pointer to handle of USB device.

Synopsis

```
typedef void *_usb_device_handle;
```

6.1.2 PTR_USB_EVENT_STRUCT

This structure is passed as a parameter to the service callback function and contains information about the event.

Synopsis

```
typedef struct _USB_EVENT_STRUCT
{
    _usb_device_handle      handle;
    uint8_t                 ep_num;
    bool                    setup;
    bool                    direction;
    uint8_t *               buffer_ptr;
    uint32_t                len;
}USB_EVENT_STRUCT, *PTR_USB_EVENT_STRUCT;
```

Fields

handle - USB control device handle

ep_num - USB endpoint number

setup - buffer_ptr contains setup packet or not

direction - direction of the endpoint, one of:

USB_RECV

USB_SEND

buffer_ptr - transferring the data buffer

len- size of the data buffer

6.1.3 USB_EP_STRUCT_PTR

This structure defines parameters that are passed to **_usb_device_init_endpoint()** API function to initialize a particular endpoint.

Synopsis

```
typedef struct _USB_EP_STRUCT
{
    uint8_t ep_num;
    uint8_t type;
    uint8_t direction;
    uint32_t size;
}USB_EP_STRUCT;
typedef USB_EP_STRUCT* USB_EP_STRUCT_PTR;
```

Fields

ep_num - USB endpoint number

type - Type of endpoint, one of:

USB_BULK_PIPE

USB_CONTROL_PIPE

USB_INTERRUPT_PIPE

direction - Direction of endpoint, one of:

USB_RECV

USB_SEND

size - Size of buffer to be used

6.2 Common Data Structures for USB Class listings

6.2.1 DESC_CALLBACK_FUNCTIONS_STRUCT

This structure is used to represent descriptor callback functions to be implemented by the application.

Synopsis

```
typedef struct _usb_desc_callbackFunction_struct
{
    uint32_t handle;
    uint8_t (_CODE_PTR_ GET_DESC)(uint32_t handle,uint8_t type,uint8_t str_num,
        uint16_t index,uint8_t_ptr *descriptor,uint32_t *size);
    USB_ENDPOINTS * (_CODE_PTR_ GET_DESC_ENDPOINTS)(uint32_t handle);
    uint8_t (_CODE_PTR_ GET_DESC_INTERFACE)(uint32_t handle,uint8_t interface,
        uint8_t_ptr alt_interface);
    uint8_t (_CODE_PTR_ SET_DESC_INTERFACE)(uint32_t handle,uint8_t interface,
        uint8_t alt_interface);
    bool (_CODE_PTR_ IS_DESC_VALID_CONFIGURATION)(uint32_t handle,
        uint16_t config_val);
    bool (_CODE_PTR_ DESC_REMOTE_WAKEUP)(uint32_t handle);
    uint8_t (_CODE_PTR_ DESC_SET_FEATURE)(uint32_t handle,int32_t cmd,
        uint8_t in_data,uint8_t ** feature);
    uint8_t (_CODE_PTR_ DESC_GET_FEATURE)(uint32_t handle,int32_t cmd,
        uint8_t in_data,uint8_t ** feature);
}DESC_CALLBACK_FUNCTIONS_STRUCT, * DESC_CALLBACK_FUNCTIONS_STRUCT_PTR;
```

Fields

handle - USB device handle

GET_DESC - The callback function is used to get various descriptors from the application.

GET_DESC_ENDPOINTS - The callback function is used to get the endpoints used and their properties.

GET_DESC_INTERFACE - The callback function is used to get the current configured interface.
SET_DESC_INTERFACE - The callback function is used to set new interface.
IS_DESC_VALID_CONFIGURATION - The callback function is used to check if the configuration is valid.

DESC_REMOTE_WAKEUP - The callback function is used to check whether the application supports remote wake-up or not.

DESC_SET_FEATURE - The callback function is used to set specific feature of the device.

DESC_GET_FEATURE - The callback function is used to get specific feature of the device.

6.2.2 USB_CLASS_CALLBACK()

This callback function is called for generic application events. The data parameter passed to the function contains information about the event. The information passed though the data parameter is based on the type of event. The application implements this callback typecasts the data parameter to the data type or structure based on the type of the event before reading it.

Synopsis

```
typedef void(_CODE_PTR_ USB_CLASS_CALLBACK)
(
    uint8_t controller_ID,
    uint8_t type,
    void*   data
);
```

Fields

controller_ID - USB controller handle

type - Type of event

data - Event data based on the type value

6.2.3 USB_CLASS_CALLBACK_STRUCT

This structure represents the class callback.

Synopsis

```
typedef struct usb_class_callback_struct
{
    USB_CLASS_CALLBACK  callback;
    void               *arg;
}USB_CLASS_CALLBACK_STRUCT, * USB_CLASS_CALLBACK_STRUCT_PTR ;
```

Fields

callback - pointer to the class callback function

arg - argument pointer to be passed in class callback function

See also:

USB_CLASS_CALLBACK()

6.2.4 USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT

This structure represents the class specific USB callback.

Synopsis

```
typedef struct usb_class_specific_handler_callback_struct
{
    USB_CLASS_SPECIFIC_HANDLER_FUNC  callback;
    void                            *arg;
}USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT,
 * USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT_PTR;
```

Fields

callback - pointer to the class callback function

arg - argument pointer to be passed in class callback function

See also:

USB_CLASS_SPECIFIC_HANDLER_FUNC()

6.2.5 USB_CLASS_SPECIFIC_HANDLER_FUNC()

This callback function supports class specific USB functionality. This function is passed as a parameter from the application to the class driver at initialization time. The parameters passed to it include request and value that the USB host sends to the device as part of the setup packet. If the application has to reply with information, it sets the data in the buffer parameter passed to it with the size information. The size parameter is an input and an output parameter that states the maximum data an application must reply with.

Synopsis

```
typedef uint8_t (_CODE_PTR_ USB_CLASS_SPECIFIC_HANDLER_FUNC)
(
    uint8_t        request,
    uint16_t       value,
    uint8_t *      *buff,
    uint32_t       *size
);
```

If a class specific request is not supported, the application passes NULL for this callback function while initializing the class layer.

Fields

request - Request code from setup packet

value - Value code from setup packet

buff - Pointer to the buffer to be returned with data

size - Size of data required from application and data sent by application

6.2.6 USB_ENDPOINTS

This structure defines information about the non-control endpoints used by the application.

Synopsis

```
typedef struct _USB_ENDPOINTS
{
    uint8_t                 count;
    USB_EP_STRUCT           *ep;
}USB_ENDPOINTS;
```

Fields

count - Count of non-control endpoints

ep - Properties of each endpoint

See also:

USB_EP_STRUCT_PTR

6.2.7 USB_REQ_CALLBACK_STRUCT

Structure other request class callback

Synopsis

```
typedef struct usb_req_callback_struct
{
    USB_REQ_FUNC            callback;
    void                    *arg;
}USB_REQ_CALLBACK_STRUCT, * USB_REQ_CALLBACK_STRUCT_PTR ;
```

Fields

ep_num - USB endpoint number

size - Size of buffer to be used in the device layer

See also:

USB_REQ_FUNC()

6.2.8 USB_REQ_FUNC()

This callback function is called to support vendor specific USB functionality and is passed from the application to the class driver at initialization time. USB control setup packet is passed to it as an input and the application returns data and size as part of the buffer as well as size output parameters passed to it.

Synopsis

```
typedef uint8_t (_CODE_PTR_ USB_REQ_FUNC)
(
    USB_SETUP_STRUCT        *setup_packet,
    uint8_t *               *buff,
    uint32_t                *size,
    void                    *arg
);
```

Fields

setup_packet — setup packet received on control endpoint from the host

buff — pointer to the buffer to be returned with data

size — size of data required from application and data sent by application

arg - other parameter

6.3 CDC Class Data Structures listings

6.3.1 CDC_HANDLE

This data type represents the CDC class handle.

Synopsis

```
typedef uint32_t CDC_HANDLE;
```

6.3.2 _ip_address

This data type represents the ip address.

Synopsis

```
typedef uint32_t _ip_address;
```

6.3.3 APP_DATA_STRUCT

This structure holds information of an endpoint buffer.

Synopsis

```
typedef struct _app_data_struct
{
    uint8_t *         data_ptr;
    uint32_t          data_size;
}APP_DATA_STRUCT;
```

Fields

data_ptr - pointer to buffer

data_size - buffer size

6.3.4 USB_CLASS_CDC_QUEUE

This structure describes a request in the endpoint queue.

Synopsis

```
typedef struct _usb_class_cdc_queue
{
    _usb_device_handle      handle;
    uint8_t                 channel;
    APP_DATA_STRUCT         app_data;
}USB_CLASS_CDC_QUEUE, *PTR_USB_CLASS_CDC_QUEUE;
```

Fields

handle - handle of USB device

channel- endpoint number of this request

app_data - endpoint buffer

See also:

APP_DATA_STRUCT

6.3.5 USB_CLASS_CDC_ENDPOINT

This structure describes an endpoint of the CDC class.

Synopsis

```
typedef struct _usb_class_cdc_endpoint
{
    uint8_t                 endpoint;
    uint8_t                 type;
    uint8_t                 bin_consumer;
    uint8_t                 bin_producer;
    USB_CLASS_CDC_QUEUE     queue[CDC_MAX_QUEUE_ELEMS];
}USB_CLASS_CDC_ENDPOINT;
```

Fields

endpoint - endpoint number

type- type of endpoint

USB_BULK_PIPE

USB_ISOCHRONOUS_PIPE

USB_BULK_PIPE

USB_INTERRUPT_PIPE

bin_consumer - the number of queued elements

bin_producer - the number of de-queued elements

queue - queue data

See also:

USB_CLASS_CDC_QUEUE

6.3.6 CDC_DEVICE_STRUCT

This structure holds CDC class state information (CDC device handle).

Synopsis

```
typedef struct _cdc_variable_struct
{
    CDC_HANDLE                  cdc_handle;
    USB_CLASS_HANDLE            class_handle;
    _usb_device_handle          controller_handle;
    USB_ENDPOINTS               *usb_ep_data;
    uint32_t                    comm_feature_data_size;
    uint8_t                     cic_send_endpoint;
    uint8_t                     cic_recv_endpoint;
    uint8_t                     dic_send_endpoint;
    uint8_t                     dic_recv_endpoint;
    uint32_t                    dic_recv_pkt_size;
    uint32_t                    dic_send_pkt_size;
    uint32_t                    cic_send_pkt_size;
    void                        *pstn_obj_ptr;
    uint8_t                     max_supported_interfaces;
    USB_CLASS_CALLBACK_STRUCT   cdc_class_cb;
    USB_REQ_CALLBACK_STRUCT      vendor_req_callback;
    USB_CLASS_CALLBACK_STRUCT   param_callback;
    USB_CLASS_CDC_ENDPOINT      *ep;
    #if RNDIS_SUPPORT
    _enet_address               mac_address;
    _ip_address                 ip_address;
    uint32_t                    rndis_max_frame_size;
    #endif
}CDC_DEVICE_STRUCT,  * CDC_DEVICE_STRUCT_PTR;
```

Fields

cdc_handle - CDC class handle

class_handle - USB common class handle

controller_handle - USB device controller handle

comm_feature_data_size - data size of communication feature

cic_send_endpoint - out notification endpoint number

cic_recv_endpoint - in notification endpoint number

dci_send_endpoint - bulk data in endpoint number

dci_recv_endpoint - bulk data out endpoint number

dic_recv_pkt_size - size of data to be received in bulk data in endpoint

dic_send_pkt_size - size of data to be sent in bulk data out endpoint

cic_send_pkt_size - size of data to be sent in notification endpoint

pstn_obj_ptr - pointer to an object of PSTN (Public Switched Telephone Network) device

max_supported_interfaces - maximum number of supported interfaces

cdc_callback - class callback function pointer

verdor_req_callback - other request class callback function pointer

param_callback - callback function pointer for application to provide class parameters

ep - pointer to the USB class MSC endpoint data

See also:

CDC_HANDLE

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_CDC_ENDPOINT

6.3.7 CDC_CONFIG_STRUCT

This structure holds configuration parameter sent by the application to configure the CDC class.

Synopsis

```
typedef struct _cdc_config_struct
{
    uint32_t                    comm_feature_data_size;
    uint8_t                     cic_send_endpoint;
    uint8_t                     dic_send_endpoint;
    uint8_t                     dic_recv_endpoint;
    uint32_t                    dic_recv_pkt_size;
    uint32_t                    dic_send_pkt_size;
    uint32_t                    cic_send_pkt_size;
    uint8_t                     max_supported_interfaces;
    USB_ENDPOINTS               *usb_ep_data;
    uint32_t                    desc_endpoint_cnt;
    USB_CLASS_CALLBACK_STRUCT   cdc_class_cb;
    USB_REQ_CALLBACK_STRUCT     vendor_req_callback;
    USB_CLASS_CALLBACK_STRUCT   param_callback;
    USB_CLASS_CDC_ENDPOINT      *ep;
    DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr;
    #if RNDIS_SUPPORT
    _enet_address               mac_address;
    _ip_address                 ip_address;
    uint32_t                    rndis_max_frame_size;
    #endif
 }CDC_CONFIG_STRUCT, * CDC_CONFIG_STRUCT_PTR;
```

Fields

comm_feature_data_size - data size of communication feature

cic_send_endpoint - out notification endpoint number

cic_recv_endpoint - in notification endpoint number

dci_send_endpoint - bulk data in endpoint number
dci_recv_endpoint - bulk data out endpoint number
dic_recv_pkt_size - size of data to be received in bulk data in endpoint
dic_send_pkt_size - size of data to be sent in bulk data out endpoint
cic_send_pkt_size - size of data to be sent in notification endpoint
max_supported_interfaces - maximum number of supported interfaces
usb_ep_data - contains all endpoints used by this device
cdc_class_callback - class callback function pointer
verdor_req_callback - other request class callback function pointer
param_callback - callback function pointer for the application to provide class parameters
ep - pointer to USB class CDC endpoint data
dec_callback_ptr - pointer to a descriptor callback function defined in the application

See also:

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_CDC_ENDPOINT

DESC_CALLBACK_FUNCTIONS_STRUCT

6.4 HID Class Data Structures listings

6.4.1 HID_HANDLE

This data type represents HID class handle.

Synopsis

```
typedef uint32_t HID_HANDLE;
```

6.4.2 USB_CLASS_HID_QUEUE

This structure describes a request in the endpoint queue.

Synopsis

```
typedef struct _usb_class_hid_queue
{
    _usb_device_handle      handle;
    uint8_t                 channel;
    uint8_t *               app_buff;
    uint32_t                size;
}USB_CLASS_HID_QUEUE, *PTR_USB_CLASS_HID_QUEUE;
```

Fields

handle - handle of USB device

channel- endpoint number of this request

app_buff - buffer to send

size - size of the transfer

6.4.3 USB_CLASS_HID_ENDPOINT

This structure contains USB class HID endpoint data.

Synopsis

```
typedef struct _usb_class_hid_endpoint
{
    uint8_t                     endpoint;
    uint8_t                     type;
    uint8_t                     bin_consumer;
    uint8_t                     bin_producer;
    USB_CLASS_HID_QUEUE         queue[HID_MAX_QUEUE_ELEMS];
}USB_CLASS_HID_ENDPOINT;
```

Fields

endpoint - endpoint number

type- type of endpoint

USB_BULK_PIPE

USB_ISOCHRONOUS_PIPE

USB_BULK_PIPE

USB_INTERRUPT_PIPE

bin_consumer - the number of queued elements

bin_producer - the number of de-queued elements

queue - queue data

See also:

USB_CLASS_HID_QUEUE

6.4.4 USB_CLASS_HID_ENDPOINT_DATA

This structure represents the endpoint data for non control endpoints.

Synopsis

```
typedef struct _usb_class_hid_endpoint_data
{
    uint8_t                         count;
    USB_CLASS_HID_ENDPOINT          *ep;
}USB_CLASS_HID_ENDPOINT_DATA, *PTR_USB_CLASS_HID_ENDPOINT_DATA;
```

Fields

count - number of non control endpoints

ep - endpoint data

See also:

USB_CLASS_HID_ENDPOINT

6.4.5 HID_DEVICE_STRUCT

This structure holds HID class state information (CDC device handle).

Synopsis

```
typedef struct hid_device_struct
{
    _usb_device_handle              handle;
    uint32_t                        user_handle;
    USB_CLASS_HANDLE                class_handle;
    USB_ENDPOINTS                   *ep_desc_data;
    USB_CLASS_CALLBACK_STRUCT       hid_class_callback;
    USB_REQ_CALLBACK_STRUCT         vendor_req_callback;
    USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT param_callback;
    USB_CLASS_HID_ENDPOINT_DATA     hid_endpoint_data;
    uint8_t                         class_request_params[2];
}HID_DEVICE_STRUCT,  * HID_DEVICE_STRUCT_PTR;
```

Fields

handle - controller device handle

user_handle - user handle
class_handle - USB class handle
ep_desc_data - contains all endpoints used by this device
hid_class_callback - class callback function pointer
verdor_req_callback - other request class callback function pointer
param_callback - callback function pointer for the application to provide class parameters
hid_endpoint_data - the endpoint data for non control endpoints
class_request_param - class request parameter for get/set idle and protocol requests

See also:

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT

USB_CLASS_HID_ENDPOINT_DATA

6.4.6 HID_CONFIG_STRUCT

This structure holds a configuration parameter sent by the application to configure the HID class.

Synopsis

```
typedef struct hid_config_struct
{
    uint32_t                        desc_endpoint_cnt;
    USB_ENDPOINTS                   *ep_desc_data;
    USB_CLASS_HID_ENDPOINT          *ep;
    USB_CLASS_CALLBACK_STRUCT       hid_class_callback;
    USB_REQ_CALLBACK_STRUCT         vendor_req_callback;
    USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT param_callback;
    DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr;
}HID_CONFIG_STRUCT, * HID_CONFIG_STRUCT_PTR;
```

Fields

desc_endpoint_cnt - number of endpoints
ep_desc_data - contains all endpoints used by this device
hid_class_callback - class callback function pointer
verdor_req_callback - other request class callback function pointer
param_callback - callback function pointer for application to provide class parameters
desc_callback_ptr - pointer to a descriptor callback function defined in the application

See also:

USB_ENDPOINTS

6.5 MSC Class Data Structures listings

6.5.1 MSD_HANDLE

This data type represents MSD class handle.

Synopsis

```
typedef uint32_t MSD_HANDLE;
```

6.5.2 APP_DATA_STRUCT

This structure holds information of an endpoint buffer.

Synopsis

```
typedef struct _app_data_struct
{
    uint8_t * data_ptr;
    uint32_t data_size;
}APP_DATA_STRUCT;
```

Fields

data_ptr - pointer to buffer

data_size - buffer size

6.5.3 USB_CLASS_MSC_QUEUE

This structure describes a request in the endpoint queue.

Synopsis

```
typedef struct _usb_class_msc_queue
{
    _usb_device_handle          handle;
    uint8_t                     channel;
    APP_DATA_STRUCT             app_data;
}USB_CLASS_MSC_QUEUE, *PTR_USB_CLASS_MSC_QUEUE;
```

Fields

handle - handle of USB device

channel- endpoint number of this request

app_data - endpoint buffer

See also:

APP_DATA_STRUCT

6.5.4 USB_CLASS_MSC_ENDPOINT

This structure describes an endpoint of the MSC class.

Synopsis

```
typedef struct _usb_class_msc_endpoint
{
    uint8_t                 endpoint;
    uint8_t                 type;
    uint8_t                 bin_consumer;
    uint8_t                 bin_producer;
    USB_CLASS_MSC_QUEUE     queue[MSD_MAX_QUEUE_ELEMS];
}USB_CLASS_MSC_ENDPOINT;
```

Fields

endpoint - endpoint number

type- type of endpoint

USB_BULK_PIPE

USB_ISOCHRONOUS_PIPE

USB_BULK_PIPE

USB_INTERRUPT_PIPE

bin_consumer - the number of queued elements

bin_producer - the number of de-queued elements

queue - queue data

See also:

USB_CLASS_MSC_QUEUE

6.5.5 LBA_APP_STRUCT

This structure holds a device logical block information.

Synopsis

```
typedef struct _lba_app_struct
{
    uint32_t                offset;
    uint32_t                size;
    uint8_t *               buff_ptr;
}LBA_APP_STRUCT, * PTR_LBA_APP_STRUCT;
```

Fields

offset - offset address of the logical block

size - size of the logical block

buff_ptr - logical block data

6.5.6 MSD_BUFF_INFO

This structure holds information of MSD buffers.

Synopsis

```
typedef struct _msd_buffers_info
{
    uint8_t *              msc_lba_send_ptr;
    uint8_t *              msc_lba_recv_ptr;
    uint32_t               msc_lba_send_buff_size;
    uint32_t               msc_lba_recv_buff_size;
}MSD_BUFF_INFO, *PTR_MSD_BUFF_INFO;
```

Fields

msc_lba_send_ptr - send buffer pointer

msc_lba_recv_ptr - receive buffer pointer

msc_lba_send_buffer_size - size of send buffer

msc_lba_recv_buffer_size - size of receive buffer

6.5.7 MSC_DEVICE_STRUCT

This structure holds MSC class state information (MSC device handle).

Synopsis

```
typedef struct _msc_variable_struct
{
    _usb_device_handle           controller_handle;
    MSD_HANDLE                   msc_handle;
    USB_CLASS_HANDLE             class_handle;
    USB_ENDPOINTS                *ep_desc_data;
    USB_CLASS_CALLBACK_STRUCT    msc_callback;
    USB_REQ_CALLBACK_STRUCT       vendor_callback;
    USB_CLASS_CALLBACK_STRUCT    param_callback;
    uint8_t                      bulk_in_endpoint;
    uint32_t                     bulk_in_endpoint_packet_size;
    uint8_t                      bulk_out_endpoint;
    uint32_t                     usb_max_suported_interfaces;
    void                         *scsi_object_ptr;
    USB_CLASS_MSC_ENDPOINT       *ep;
    uint8_t                      lun;
    bool                         out_flag;
    bool                         in_flag;
    bool                         in_stall_flag;
    bool                         out_stall_flag;
    bool                         cbw_valid_flag;
    PTR_CSW                      csw_ptr;
    PTR_CBW                      cbw_ptr;
    bool                         re_stall_flag;
    DEVICE_LBA_INFO_STRUCT       device_info;
    MSD_BUFF_INFO                msd_buff;
```

```
    uint32_t                    transfer_remaining;
    uint32_t                    current_offset;
}MSC_DEVICE_STRUCT,  * MSC_DEVICE_STRUCT_PTR;
```

Fields

controller_handle - device controller handle

msc_handle - MSC class handle

class_handle - USB common class handle

ep_desc_data - contains all endpoints used by this device

msc_callback - class callback function pointer

verdor_req_callback - other request class callback function pointer

param_callback - callback function pointer for application to provide class parameters

bulk_in_endpoint - receive bulk endpoint

bulk_in_endpoint_packet_size - size of receive bulk endpoint

bulk_out_endpoint - send bulk endpoint

usb_max_suported_interfaces - maximum number of supported interfaces

scsi_object_ptr - pointer to SCSI object

ep - pointer to USB class MSC endpoint data

lun - logical unit number. It can have the value only from 0 to 15 decimals

out_flag - flag to track bulk out data processing after command block wrapper if needed

in_flag - flag to track bulk in data processing before command status wrapper if needed

in_stall_flag - flag to track if there is need to stall BULK IN ENDPOINT because of BULK COMMAND

out_stall_flag - flag to track if there is need to stall BULK OUT ENDPOINT because of BULK COMMAND

cbw_valid_flag - flag to validate command block wrapper

csw_ptr - global structure for command status wrapper

cbw_ptr global structure for command block wrapper

re_stall_flag - re-installation flag

device_info - device information

msd_buff - contain information of msd class buffers

transfer_remaining - number of remaining transfer bytes

current_offset - offset of remaining transfer bytes

See also:

MSD_HANDLE

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_MSC_ENDPOINT

LBA_APP_STRUCT

MSD_BUFF_INFO

6.5.8 USB_MSD_CONFIG_STRUCT

This structure holds configuration parameter sent by application to configure the MSC class.

Synopsis

```
typedef struct _usb_msd_config
{
    DEVICE_LBA_INFO_STRUCT        device_info;
    bool                          implementing_disk_drive;
    uint32_t                      usb_max_suported_interfaces;
    uint8_t                       bulk_in_endpoint;
    uint32_t                      bulk_in_endpoint_packet_size;
    uint8_t                       bulk_out_endpoint;
    uint32_t                      desc_endpoint_cnt;
    MSD_BUFF_INFO                 msd_buff;
    USB_ENDPOINTS                 *ep_desc_data;
    USB_CLASS_MSC_ENDPOINT        *ep;
    USB_CLASS_CALLBACK_STRUCT_PTR msc_class_callback;
    USB_REQ_CALLBACK_STRUCT_PTR   vendor_req_callback;
    USB_CLASS_CALLBACK_STRUCT_PTR param_callback;
    DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr;
}USB_MSD_CONFIG_STRUCT,  * USB_MSD_CONFIG_STRUCT_PTR;
```

Fields

device_info - device information

implementing_disk_drive - If Implementing Disk Drive then configure the macro below as TRUE, otherwise keep it FALSE (say for Hard Disk)

usb_max_suported_interfaces - maximum number of supported interfaces

bulk_in_endpoint - receive bulk endpoint

bulk_in_endpoint_packet_size - size of receive bulk endpoint

bulk_out_endpoint - send bulk endpoint

usb_max_suported_interfaces - maximum number of supported interfaces

msd_buff - contain information of MSC class buffers

ep_desc_data - contains all endpoints used by this device

ep - pointer to USB class MSC endpoint data

msc_callback - class callback function pointer

verdor_req_callback - other request class callback function pointer

param_callback - callback function pointer for application to provide class parameters

desc_callback_ptr - pointer to descriptor callback function defined in application

See also:

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_MSC_ENDPOINT

DESC_CALLBACK_FUNCTIONS_STRUCT

MSD_BUFF_INFO

6.6 PHDC Class Data Structures listings

6.6.1 PHDC_HANDLE

This data type represents PHDC class handle.

Synopsis

```
typedef uint32_t PHDC_HANDLE;
```

6.6.2 USB_CLASS_PHDC_QOS_BIN

This structure holds a request in the endpoint QOS bin.

Synopsis

```
struct _usb_class_phdc_qos_bin
{
    uint8_t                 channel;
    bool                    meta_data;
    uint8_t                 num_tfr;
    uint8_t                 qos;
    uint8_t *               app_buff;
    uint32_t                size;
};
typedef struct _usb_class_phdc_qos_bin USB_CLASS_PHDC_QOS_BIN,
*PTR_USB_CLASS_PHDC_QOS_BIN;
```

Fields

channel- endpoint number of this request

meta_data - whether a packet is a meta data or not

num_tfr - number of transfers that follow the meta data package. Used only when meta_data is TRUE

qos - quality of the transfers that follow the meta data package

app_buff - buffer to send

size - size of the transfer

6.6.3 USB_CLASS_PHDC_TX_ENDPOINT

This structure holds transmission endpoint data information of the PHDC class.

Synopsis

```
typedef struct _usb_class_phdc_tx_endpoint
{
uint8_t                     endpoint;
uint8_t                     type;
uint32_t                    size;
uint8_t                     qos;
uint8_t                     current_qos;
uint8_t                     transfers_left;
```

```
uint8_t                     bin_consumer;
uint8_t                     bin_producer;
USB_CLASS_PHDC_QOS_BIN      qos_bin[MAX_QOS_BIN_ELEMS];
}USB_CLASS_PHDC_TX_ENDPOINT;
```

Fields

endpoint - endpoint number

type- type of endpoint

USB_BULK_PIPE

USB_ISOCHRONOUS_PIPE

USB_BULK_PIPE

USB_INTERRUPT_PIPE

size - size of transfer

qos - quality of transfer

current_qos - quality of received meta data

transfers_left - number of transfers left

bin_consumer - the number of queued elements

bin_producer - the number of de-queued elements

qos_bin - requests in the endpoint QOS bin

See also:

USB_CLASS_PHDC_QOS_BIN

6.6.4 USB_CLASS_PHDC_RX_ENDPOINT

This structure holds receive endpoint data information of PHDC class.

Synopsis

```
typedef struct _usb_class_phdc_rx_endpoint
{
    uint8_t           endpoint;
    uint8_t           type;
    uint32_t          size;
    uint8_t           qos;
    uint8_t           current_qos;
    uint8_t           transfers_left;
    uint16_t          buffer_size;
    uint8_t *         buff_ptr;
}USB_CLASS_PHDC_RX_ENDPOINT;
```

Fields

endpoint - endpoint number

type- type of endpoint

USB_BULK_PIPE

USB_ISOCHRONOUS_PIPE

USB_BULK_PIPE

USB_INTERRUPT_PIPE

size - size of transfer

qos - quality of transfer

current_qos - quality of received meta data

transfers_left - number of transfers left

buffer_size - size of receive buffer

buff_ptr - receive buffer

6.6.5 USB_CLASS_PHDC_ENDPOINT_DATA

This structure holds endpoint information of the PHDC class.

Synopsis

```
typedef struct _usb_class_phdc_endpoint_data
{
    _usb_device_handle              handle;
    uint8_t                         count_rx;
    uint8_t                         count_tx;
    USB_CLASS_PHDC_RX_ENDPOINT      ep_rx[PHDC_RX_ENDPOINTS];
    USB_CLASS_PHDC_TX_ENDPOINT      ep_tx[PHDC_TX_ENDPOINTS];
}USB_CLASS_PHDC_ENDPOINT_DATA, *PTR_USB_CLASS_PHDC_ENDPOINT_DATA;
```

Fields

handle - device controller handle

count_rx - number of receive endpoints

count_tx - number of transmission endpoints

ep_rx - receive endpoint description

ep_tx - send endpoint description

See also:

USB_CLASS_PHDC_TX_ENDPOINT

USB_CLASS_PHDC_RX_ENDPOINT

6.6.6 USB_APP_EVENT_SEND_COMPLETE

This structure holds data passed to the application when the send process is completed.

Synopsis

```
typedef struct _usb_app_event_send_complete
{
    uint8_t             qos;
    uint8_t *           buffer_ptr;
    uint32_t            size;
}USB_APP_EVENT_SEND_COMPLETE, *PTR_USB_APP_EVENT_SEND_COMPLETE;
```

Fields

qos - quality of the transfer

buffer_ptr - send buffer pointer

size - size of buffer

6.6.7 USB_APP_EVENT_DATA_RECIEVED

This structure holds data passed to the application when the receive process is completed.

Synopsis

```
typedef struct _usb_app_event_data_recieved
{
    uint8_t           qos;
    uint8_t *         buffer_ptr;
    uint32_t          size;
}USB_APP_EVENT_DATA_RECIEVED, *PTR_USB_APP_EVENT_DATA_RECIEVED;
```

Fields

qos - quality of the transfer

buffer_ptr - send buffer pointer

size - size of buffer

6.6.8 PHDC_STRUCT

This structure holds the PHDC class state information (PHDC device handle).

Synopsis

```
typedef struct _phdc_struct
{
    _usb_device_handle            controller_handle;
    PHDC_HANDLE                   phdc_handle;
    USB_CLASS_HANDLE              class_handle;
    USB_CLASS_CALLBACK_STRUCT     phdc_callback;
    USB_REQ_CALLBACK_STRUCT       vendor_callback;
    uint8_t *                     service_buff_ptr;
    USB_CLASS_PHDC_ENDPOINT_DATA  ep_data;
    #if META_DATA_MSG_PRE_IMPLEMENTED
    USB_META_DATA_MSG_PREAMBLE    meta_data_msg_preamble;
    #endif
    #if USB_METADATA_SUPPORTED
    bool                          phdc_metadata;
    #endif
    uint16_t                      phdc_ep_has_data;
}PHDC_STRUCT,  * PHDC_STRUCT_PTR;
```

Fields

controller_handle - controller device handle

phdc_handle - PHDC class handle

class_handle - USB common class handle
phdc_class_callback - class callback function pointer
verdor_req_callback - other request class callback function pointer
service_buff_ptr - ram buffer for configuring next receive
ep_data - PHDC endpoint data
phdc_ep_has_data - stores a bit map of the active endpoints

See also:

PHDC_HANDLE

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

USB_CLASS_PHDC_ENDPOINT_DATA

6.6.9 PHDC_CONFIG_STRUCT

This structure holds configuration parameter sent by the application to configure the HID class.

Synopsis

```
typedef struct _config_phdc_struct
{
    USB_CLASS_CALLBACK_STRUCT           phdc_callback;
    USB_REQ_CALLBACK_STRUCT             vendor_callback;
    DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr;
    USB_ENDPOINTS                       *info;
}PHDC_CONFIG_STRUCT, * PHDC_CONFIG_STRUCT_PTR;
```

Fields

phdc_class_callback - class callback function pointer
verdor_req_callback - other request class callback function pointer
desc_callback_ptr - pointer to a descriptor callback function defined in the application
info - contains all endpoints used by this device

See also:

USB_ENDPOINTS

USB_CLASS_CALLBACK_STRUCT

USB_REQ_CALLBACK_STRUCT

DESC_CALLBACK_FUNCTIONS_STRUCT

Chapter 7
Reference Data Types

7.1 USB Device API Data Types

USB Device API uses the data types as shown in Table 7-1.

Table 7-1. USB Device API Data Types

USB Device API data type	Simple data type
_usb_device_handle	void*