# USB Stack Device Reference Manual

Document Number: USBSDRM

Rev. 2, 09/2015

# Contents

# Chapter 1 Before You Begin

## 1.1 About this book

This *USB Stack Device Reference Manual* describes the USB Device driver and the programming interface in the USB Stack.

The audience should be 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:

- *Source Code*

## 1.2 Acronyms and abbreviations

**Table 1 Acronyms and abbreviations**

| Term | Description |
|---|---|
| API | Application Programming Interface |
| CDC | Communication Device Class |
| HID | Human Interface Device |
| MSD | Mass Storage Device |
| MSC | Mass Storage Class |
| PHDC | Personal Healthcare Device Class |
| ZLT | Zero Length Transfer |
| LAB | Logical Address Block |
| LUN | Logical Unit Number |

## 1.3 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 –`* indicates that the function uses one or more values in the parameter you give it without storing any changes.
- *`Out –`* indicates that the function saves one or more values in the parameter you give it. Examine the saved values to find out useful information about your application.
- *`In/out`* – indicates that the function changes one or more values in the parameter you give it and saves the result. 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()**.

# Chapter 2 Overview

## 2.1 USB overview

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

The USB Device software consists of the following parts:

- USB Device application
- USB Device Class Driver (contains USB Device Class APIs)
- USB Device Common Controller Driver APIs (independent of hardware)
- USB Device controller interface (DCI) - low-level functions used to interact with the USB Device controller hardware
- OS adapter to provide unified OS API to USB Stack
- SoC-specific initialization.

  In <install_dir>/usb/usb_core/device/sources/bsp/SOC_NAME/usb_dev_bsp.c

  ```
  usb_status usb_dev_soc_init(uint8_t controller_id)
  ```

  ```
  The controller_id is the enum CONTROLLER_INDEX
  ```

  This function should be implemented by users when porting to a new platform.

- Board-specific initialization.

  In <install_dir>/examples/BOARD_NAME /board.c

  ```
  uint8_t usb_device_board_init(uint8_t controller_id)
  ```

  ```
  The controller_id is the enum CONTROLLER_INDEX
  ```

  This function must be implemented by users when porting to a new board.

**Note**

> As a result of the FS controller (KHCI) IP limitation, one endpoint can be enabled for one direction only, either IN or OUT. It can't be enabled for both directions.

The whole architecture and components of USB stack are as follows:

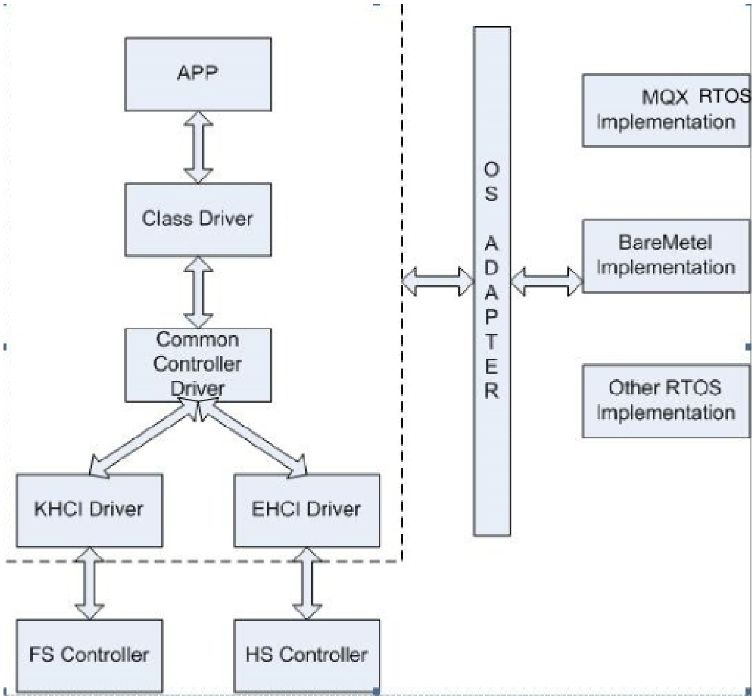

**Figure 1 USB Device stack architecture**

## 2.2 API overview

This section describes the API functions. The interfaces between the USB Common Controller driver and xHCI driver are not listed here. All USB device APIs can’t be invoked during the interrupt service routine except when the operating system is bare metal.

Table 2 summarizes the USB Common Controller driver APIs.

**Table 2 USB Device Controller driver APIs**

| No. | API Function | Description |
|---|---|---|
| 1 | usb_device_init() | Initializes the USB device controller |
| 2 | usb_device_postinit() | Some additional initialization actions need to be done after the device is initialized |
| 3 | usb_device_deinit() | Un-initializes the device controller |
| 4 | usb_device_recv_data() | Receives data from a specified endpoint |
| 5 | usb_device_send_data() | Sends data to a specified endpoint |
| 6 | usb_device_cancel_transfer() | Cancels all the pending transfers in a specified endpoint |

|  |  |  |
|---|---|---|
| 7 | usb_device_register_service() | Registers a callback function for the service |
| 8 | usb_device_unregister_service() | Unregisters a callback function for the service |
| 9 | usb_device_init_endpoint() | Initializes the specified endpoint |
| 10 | usb_device_deinit_endpoint() | Un-initializes the specified endpoint |
| 11 | usb_device_stall_endpoint() | Stalls the specified endpoint |
| 12 | usb_device_unstall_endpoint() | Un-stalls the specified endpoint |
| 13 | usb_device_register_application_notify() | Registers the callback function for the application related event |
| 14 | usb_device_register_vendor_class_request_notify() | Registers the callback function for the vendor class related event |
| 15 | usb_device_register_desc_request_notify() | Registers the callback function for the descriptor related event |
| 16 | usb_device_set_status() | Sets the current status of the selected item |
| 17 | usb_device_get_status() | Gets the current status of the selected item |

Table 3 summarizes the common class APIs.

**Table 3 Common class driver APIs**

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_Init() | Initializes the class module |
| 2 | USB_Class_Deinit() | Un-initializes the class module |
| 3 | USB_Class_Send_Data() | Sends data on the specified endpoint |

Table 4 summarizes the CDC class APIs.

**Table 4 CDC class driver APIs**

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_CDC_Init() | Initializes the CDC class |

| | | |
|---|---|---|
| 2 | USB_Class_CDC_Deinit() | Un-initializes the CDC class driver |
| 3 | USB_Class_CDC_ Recv _Data () | Receives data on the specified endpoint |
| 4 | USB_Class_CDC_Send_Data() | Sends data on the specified endpoint |
| 5 | USB_Class_CDC_Cancel() | Cancels all the uncompleted transfers in the specified endpoint |
| 6 | USB_Class_CDC_Get_Speed() | Get current USB speed of the CDC device |

Table 5 summarizes the HID class APIs.

**Table 5 HID class driver APIs**

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_HID_Init() | Initializes the HID class |
| 2 | USB_Class_HID_Deinit() | Un-initializes the HID class driver |
| 3 | USB_Class_HID_Send_Data() | Sends data on the specified endpoint |
| 4 | USB_Class_HID_Cancel() | Cancels all the uncompleted transfers in the specified endpoint |
| 5 | USB_Class_HID_Recv_Data() | Receives data on the specified endpoint |
| 6 | USB_Class_HID_Get_Speed() | Get current USB speed of the HID device |

Table 6 summarizes the MSC class APIs.

**Table 6 MSC class driver APIs**

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_MSC_Init() | Initializes the MSC class |
| 2 | USB_Class_MSC_Deinit() | Un-initializes the MSC class driver |
| 3 | USB_Class_MSC_Get_Speed() | Get current USB speed of the MSC device |

Table 7 summarizes the AUDIO class APIs.

Table 7 AUDIO class driver APIs

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_Audio_Init() | Initializes the AUDIO class |
| 2 | USB_Class_Audio_Deinit() | Un-initializes the AUDIO class driver |
| 3 | USB_Class_Audio_Recv _Data() | Receives data on the specified endpoint |
| 4 | USB_Class_Audio_Send _Data() | Sends data on the specified endpoint |
| 5 | USB_Class_Audio_Cancel() | Cancels all the uncompleted transfers in the specified endpoint |
| 6 | USB_Class_Audio_Get_Speed() | Get current USB speed of the Audio device |

Table 8 summarizes the PHDC class APIs.

Table 8 PHDC class driver APIs

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Class_PHDC_Init() | Initializes the PHDC class |
| 2 | USB_Class_PHDC_Deinit() | Un-initializes the PHDC class driver |
| 3 | USB_Class_PHDC_Recv _Data () | Receives data on the specified endpoint |
| 4 | USB_Class_PHDC_Send _Data() | Sends data on the specified endpoint |
| 5 | USB_Class_PHDC_Cancel() | Cancels all the uncompleted transfers in the specified endpoint |
| 6 | USB_Class_PHDC_Get_Speed() | Get current USB speed of the PHDC device |

Table 9 summarizes the Composite class APIs.

Table 9 Composite Class driver APIs

| No. | API Function | Description |
|---|---|---|
| 1 | USB_Composite_Init() | Initializes the Composite class |
| 2 | USB_Composite_Deinit() | Un-initializes the Composite class driver |
| 6 | USB_Composite_Get_Speed() | Get current USB speed of the Composite device |

## 2.3 Using the USB Device API

### 2.3.1 Using USB Common Controller driver API

It is not recommend using the USB Common Controller driver directly to implement the USB function, but you can see the code implemented in the class driver provided by Freescale for detailed information.

### 2.3.2 Using CDC class driver 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_DEV_EVENT_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 the callback function is called with the USB_DEV_EVENT_DATA_RECEIVED event, which implies that the previous reception of data from the host is done.

### 2.3.3 Using HID class driver 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_DEV_EVENT_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.

### 2.3.4 Using MSC class driver 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_DEV_EVENT_ENUM_COMPLETE event, the application should move into the ready state.
3. The callback function is called with the USB_MSC_DEVICE_READ_REQUEST event to get 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. The callback function is called with the USB_MSC_DEVICE_WRITE_REQUEST event to prepare the storage device buffer for USB transfer.
5. The callback function is called with the USB_DEV_EVENT_DATA_RECEIVED event to write the storage device buffer data the storage device.

### 2.3.5 Using AUDIO class driver API

To use AUDIO class layer API functions from the application:

1. Call **USB_Class_Audio_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_DEV_EVENT_ENUM_COMPLETE event, the application should move into the connected state.
3. Call **USB_Class_Audio_Send_Data()** to send data to the host through the device layers, when required.
4. Call **USB_Class_Audio_Recv_Data()** when the callback function is called with the USB_DEV_EVENT_DATA_RECEIVED event, which implies that the previous reception of data from the host is done.

### 2.3.6 Using PHDC class driver 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_DEV_EVENT_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 the callback function is called with the USB_DEV_EVENT_DATA_RECEIVED event, which implies that the previous reception of data from the host is done.

### 2.3.7 Using composite class driver API

1. Call **USB_Composite_Init()** to initialize the composite class driver, all the layers below it, and the device controller. Event callback functions for each interface are also passed as parameter to this function.
2. Read class_handle to get every class handle for each interface composited in the composited device.
3. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application should move into the connected state.
4. Call the corresponding Send API to send data to the host through the device layers, when required. For example, if the device is composited of HID and AUDIO, **USB_Class_Audio_Send_Data()** can be used to send data to the host with the handle obtained from class_handle.
5. Call the corresponding receive API when the callback function is called with the USB_DEV_EVENT_DATA_RECEIVED event, which implies that the previous reception of data from the host is done. For example, if the device is composited of HID and AUDIO, **USB_Class_Audio_Recv_Data()** can be used to send data to the host with the handle obtained from class_handle.

# Chapter 3 USB Common Controller Driver API

## 3.1 usb_device_init

Initializes the USB device controller

**Synopsis**

```
usb_status usb_device_init
(
        uint8_t                 controller_id,
        usb_device_handle*      handle
)
```

**Parameters**

| | | |
|---|---|---|
| *controller_id* | *[in]* – | controller ID, such as USB_CONTROLLER_KHCI_0 |
| *handle* | *[out]* – | USB Device handle |

**Description**

The function initializes the device controller specified by the controller_id and a device handle can be returned from the handle.

**Return value**

- **USB_OK** (success)
- **Other** (failure)

## 3.2 usb_device_postinit

Initializes the USB device controller

**Synopsis**

```
usb_status usb_device_postinit
(
        uint8_t                 controller_id,
        usb_device_handle       handle
)
```

**Parameters**

| | | |
|---|---|---|
| *controller_id* | *[in]* – | controller ID, such as USB_CONTROLLER_KHCI_0 |
| *handle* | *[in]* – | USB Device handle |

**Description**

The function starts the initialization process that cannot be done in the **usb_device_init()** API. For example, the call back functions need to be registered after the device handle can be obtained from the **usb_device_init()** API. Therefore, the USB interrupt cannot be enabled in **usb_device_init()**; otherwise, the USB interrupt can be issued before the callback functions are registered. To avoid this issue, the USB interrupt will be enabled in the post initialization process.

**Return value**

- **USB_OK** (success)
- **Other** (failure)

## 3.3 usb_device_deinit

Deinitializes the USB device controller

**Synopsis**

```
usb_status usb_device_deinit
(
        usb_device_handle      handle
)
```

**Parameters**

*handle* *[in]* – USB Device handle

**Description**

The function deinitializes the device controller specified by the handle.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.4 usb_device_recv_data

Receives data from a specified endpoint

**Synopsis**

```
usb_status usb_device_recv_data
(
usb_device_handle         handle,
uint_8              ep_index,
uint_8*             buff_ptr,
uint_32             size
)
```

**Parameters**

*handle* *[in]* – USB Device handle

*ep_index* *[in]* – endpoint index

*buff_ptr* *[in]* – memory address to receive the data

| | | |
|---|---|---|
| *size* | *[in]* | – length of the packet to be received |

**Description**

The function is used to receive data from a specified endpoint. For each endpoint, this function can only be called once, before the last transmission completed.

**Return value**

- **USB_OK** (success)
- **other** (failure)

**Note:** The return value just indicates if the receiving request is successful or not; the transfer done is notified by the corresponding callback function.

The transmission includes recv_data and send_data.

## 3.5 usb_device_send_data

Sends data from a specified endpoint

**Synopsis**

```
usb_status usb_device_send_data
(
usb_device_handle          handle,
uint_8               ep_index,
uint_8*              buff_ptr,
uint_32              size
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* | – USB Device handle |
| *ep_index* | *[in]* | – endpoint index |
| *buff_ptr* | *[in]* | – memory address hold the data need to be sent |
| *size* | *[in]* | – length of the packet to be received |

**Description**

The function is used to send data to a specified endpoint. For each endpoint, this function can only be called once, before the last transmission completed.

**Return value**

- **USB_OK** (success)
- **other** (failure)

**Note:** The return value just indicates if the sending request is successful or not and the completed transfer is notified by the corresponding callback function.

The transmission includes recv_data and send_data.

## 3.6 usb_device_cancel_transfer

Cancels all the pending transfers in a specified endpoint.

**Synopsis**

```
usb_status usb_device_cancel_transfer
(
usb_device_handle         handle,
uint_8              ep_index,
uint_8              direction

)
```

**Parameters**

*handle* [in] – USB Device handle

*ep_index* [in] – endpoint index

*direction* [in] – direction of the endpoint

**Description**

The function is used to cancel all the pending transfer in a specified endpoint which is determined by the endpoint index and the direction.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.7 usb_device_register_service

Registers a callback function for one specified endpoint.

**Synopsis**

```
usb_status usb_device_register_service
(
usb_device_handle         handle,
uint8_t             type,
usb_event_service_t  service,
void*                     arg
)
```

**Parameters**

*handle* [in] – USB Device handle

*type* *[in]* – service type, type & 0xF is the endpoint index

*service* *[in]* – callback function

*arg* *[in]* – second parameter for the callback function

**Description**

The function is used to register a callback function for one specified endpoint.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.8 usb_device_unregister_service

Unregisters a callback function for one specified endpoint.

**Synopsis**

```
usb_status usb_device_unregister_service
(
usb_device_handle         handle,
uint8_t              type
)
```

**Parameters**

*handle* *[in]* – USB Device handle

*type* *[in]* – service type, type & 0xF is the endpoint index

**Description**

The function is used to unregister a callback function for one specified endpoint.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.9 usb_device_init_endpoint

Initializes the specified endpoint.

**Synopsis**

```
usb_status usb_device_init_endpoint
(
usb_device_handle         handle,
usb_ep_struct_t*          ep_ptr,
uint_8                    flag
)
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – USB Device handle |
| *ep_ptr* | *[in]* – endpoint information, see Section 7.1.1 |
| *flag* | *[in]* – whether the ZLT is enabled for this endpoint |

**Description**

The function is used to initialize a specific endpoint which is determined by the ep_ptr.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.10 usb_device_deinit_endpoint

Un-initializes the specified endpoint.

**Synopsis**

```
usb_status usb_device_deinit_endpoint
(
usb_device_handle           handle,
uint8_t              ep_num,
uint_8                      direction
)
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – USB Device handle |
| *ep_num* | *[in]* – endpoint index |
| *direction* | *[in]* –endpoint direction |

**Description**

The function is used to un-initialize a specific endpoint which is determined by the endpoint index and endpoint direction.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.11 usb_device_stall_endpoint

Stalls the specified endpoint.

**Synopsis**

```
usb_status usb_device_stall_endpoint
(
usb_device_handle          handle,
uint8_t              ep_num,
uint_8                     direction
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* – | USB Device handle |
| *ep_num* | *[in]* – | endpoint index |
| *direction* | *[in]* – | endpoint direction |

**Description**

The function is used to stall a specific endpoint which is determined by the endpoint index and endpoint direction.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.12 usb_device_unstall_endpoint

Un-stalls the specified endpoint.

**Synopsis**

```
usb_status usb_device_unstall_endpoint
(
usb_device_handle          handle,
uint8_t              ep_num,
uint_8                     direction
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* – | USB Device handle |
| *ep_num* | *[in]* – | endpoint index |
| *direction* | *[in]* – | endpoint direction |

**Description**

The function is used to un-stall a specific endpoint which is determined by the endpoint index and endpoint direction.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.13 usb_device_register_application_notify

Registers the callback function for the application related event.

**Synopsis**

```
usb_status usb_device_register_application_notify
(
        usb_device_handle       handle,
         usb_device_notify_t    device_notify_callback,
        void*                   device_notify_param
)
```

**Parameters**

*handle* *[in]* – USB Device handle

*device_notify_callback* *[in]* – callback function

*device_notify_param* *[in]* – parameter for the callback function

**Description**

The function is used to register a callback function for the application related event. Currently the following events are supported:

| Event | Description |
|---|---|
| USB_DEV_EVENT_BUS_RESET | A BUS reset is received. |
| USB_DEV_EVENT_ENUM_COMPLETE | The device enumerated process completes. |
| USB_DEV_EVENT_CONFIG_CHANGED | Host sends a set_configuration. |
| USB_DEV_EVENT_ERROR | Error. |

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.14 usb_device_register_vendor_class_request_notify

Registers the callback function for the vendor class related event.

**Synopsis**

```
usb_status usb_device_register_vendor_class_request_notify
(
        usb_device_handle       handle,
         usb_request_notify_t    request_notify_callback,
        void*                   request_notify_param
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* | – USB Device handle |
| *request_notify_callback* | *[in]* | – callback function |
| *request_notify_param* | *[in]* | – parameter for the callback function |

**Description**

The function is used to register a callback function for the vendor class request related event. Currently the vendor class is not implemented, so both the **request_notify_callback** and **request_notify_param** can be set to NULL.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.15 usb_device_register_desc_request_notify

Registers the callback functions for the device descriptor related request.

**Synopsis**

```
usb_status usb_device_register_desc_request_notify
(
        usb_device_handle       handle,
        usb_desc_request_notify_struct_t*   desc_request_notify_callback,
        void*                   desc_request_notify_param
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* | – USB Device handle |
| *desc_request_notify_callback* | *[in]* | – callback function |
| *desc_request_notify_param* | *[in]* | – parameter for the callback function |

**Description**

The function is used to register a set of callback functions for the device descriptor related event. For details, see Section 7.1.7.

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.16 usb_device_get_status

Gets the internal USB device state.

**Synopsis**

```
usb_status usb_device_get_status
(
usb_device_handle        handle,
uint8_t             component,
uint16_t*           status
)
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – USB Device handle |
| *component* | *[in]* – callback function |
| *status* | *[out]* – requested status |

**Description**

The function is used to get the status of the specified component. The supported components include:

- USB_STATUS_DEVICE_STATE
- USB_STATUS_OTG
- USB_STATUS_DEVICE
- USB_STATUS_ENDPOINT, the LSB nibble carries the endpoint number

**Return value**

- **USB_OK** (success)
- **other** (failure)

## 3.17 usb_device_set_status

Sets the internal USB device state.

**Synopsis**

```
usb_status usb_device_set_status
(
usb_device_handle        handle,
uint8_t                  component,
uint16_t                 status
)
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – USB Device handle |
| *component* | *[in]* – callback function |
| *status* | *[in]* – status to set |

**Description**

The function is used to set the status of the specified component. The supported components include:

- USB_STATUS_DEVICE_STATE
- USB_STATUS_OTG
- USB_STATUS_DEVICE

**Return value**

- **USB_OK** (success)
- **other** (failure)

# Chapter 4 USB Device Class API

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

## 4.1 CDC class API function listings

### 4.1.1 USB_Class_CDC_Init()

Initializes the CDC class.

**Synopsis**

```
usb_status USB_Class_CDC_Init
(
            uint8_t                 controller_id,
            cdc_config_struct_t*    cdc_config_ptr,
            cdc_handle_t*           cdc_handle_ptr
)
```

**Parameters**

*controller_id* *[in]* – controller ID, such as USB_CONTROLLER_KHCI_0

*cdc_config_ptr* *[in]* – CDC configuration structure, see **cdc_config_struct_t**

**Note**

In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

*cdc_handle_ptr* *[out]* – pointer point to the initialized CDC class, see **cdc_handle_t**

**Description**

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

**Return Value**

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

### 4.1.2 USB_Class_CDC_Deinit()

Un-initializes the CDC class.

**Synopsis**

```
usb_status USB_Class_CDC_Deinit
(
            cdc_handle_t            cdc_handle
    )
```

**Parameters**

| | |
|---|---|
| `cdc_handle` | `[in]` – The CDC class handler |

**Description**

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

**Return Value**

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

### 4.1.3 USB_Class_CDC_Recv_Data()

Receive the CDC data.

**Synopsis**

```
usb_status USB_Class_CDC_Recv_Data

(

            cdc_handle_t            cdc_handle,

            uint8_t                 ep_num,

            uint8_t*                app_buff,

            uint32_t                size

)
```

**Parameters**

| | |
|---|---|
| *cdc_handle* | *[in]* – CDC class handler |
| *ep_num* | *[in]* – endpoint number |
| *app_buff* | *[in]* – buffer to save the data from the host |
| *size* | *[in]* – buffer length to receive |

**Description**

The application calls this API function to receive data from the host. Once the data is received, the application layer receives a callback event USB_DEV_EVENT_DATA_RECEIVED. The transfer failed when the value of the object that the pointer size points is 0xFFFFFFFF in callback event USB_DEV_EVENT_DATA_RECEIVED.

**Return Value**

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

### 4.1.4 USB_Class_CDC_Send_Data()

Sends the CDC data.

**Synopsis**

```
usb_status USB_Class_CDC_Send_Data
(
             cdc_handle_t       cdc_handle,
             uint8_t                  ep_num,
              uint8_t*                app_buff,
              uint32_t                size
)
```

**Parameters**

| | |
|---|---|
| *cdc_handle* | *[in]* – CDC class handler |
| *ep_num* | *[in]* – endpoint number |
| *app_buff* | *[in]* – buffer to send |
| *size* | *[in]* – buffer length to send |

**Description**

The application calls this API function to send DIC data specified by *app_buff* and *size*. Data is sent through DIC_BULK_IN_ENDPOINT. Once the data is sent, the application layer receives a callback event USB_DEV_EVENT_SEND_COMPLETE. The transfer failed when the value of the object that the pointer size points is 0xFFFFFFFF in callback event USB_DEV_EVENT_SEND_COMPLETE. The application reserves the buffer until it receives a callback event indicating that the data is sent.

**Return Value**

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

### 4.1.5 USB_Class_CDC_Cancel()

Cancels all the uncompleted transfers in the specified endpoint.

**Synopsis**

```
usb_status USB_Class_CDC_Cancel
(
        cdc_handle_t            cdc_handle,
        uint8_t                 ep_num
)
```

**Parameters**

| | |
|---|---|
| cdc_handle | [in] – CDC class handler |
| ep_num | [in] – endpoint number |

**Description**

The application calls this API function to cancel all the uncompleted transfers in the specified endpoint.

**Return Value**

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

### 4.1.6 USB_Class_CDC_Get_Speed()

Get the USB speed of the CDC device.

**Synopsis**

```
usb_status USB_Class_CDC_Get_Speed
(
         cdc_handle_t          cdc_handle,
         uint16_t *            speed
)
```

**Parameters**

| | |
|---|---|
| cdc_handle | [in] – CDC class handler |
| speed | [out] – current speed |

**Description**

The application calls this API function to get the USB speed of the CDC device.

**Return Value**

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

## 4.2 HID class API function listings

### 4.2.1 USB_Class_HID_Init()

Initializes the HID class.

**Synopsis**

```
usb_status USB_Class_HID_Init
(
              uint8_t                  controller_id,
                 hid_config_struct_t*    hid_config_ptr,
                 hid_handle_t*           hidHandle
    )
```

**Parameters**

*`controller_id`* *`[in]`* – controller ID, such as USB_CONTROLLER_KHCI_0

*`hid_config_ptr`* *`[in]`* – HID configuration structure, see `hid_config_struct_t`

**Note**

In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

*`hidHandle`* *`[out]`* – pointer point to the initialized HID class, see hid_handle_t

**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)

### 4.2.2 USB_Class_HID_Deinit()

Un-initializes the HID class.

**Synopsis**

```
    usb_status USB_Class_HID_Deinit
    (
            hid_handle_t      handle
    )
```

**Parameters**

*`handle`* *`[in]`* – HID class handler

**Description**

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

**Return Value**

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

### 4.2.3 USB_Class_HID_Send_Data()

Sends the HID data.

**Synopsis**

```
usb_status USB_Class_HID_Send_Data
(
               hid_handle_t            handle,
                uint8_t                ep_num,
               uint8_t*                app_buff,
               uint32_t                size
)
```

**Parameters**

*handle* *[in]* – HID class handler

*ep_num* *[in]* – endpoint number

*app_buff* *[in]* – buffer to send

*size* *[in]* – buffer length to send

**Description**

The application calls this API function to send HID data specified by *app_buff* and *size*. Once the data is sent, the application layer receives a callback event USB_DEV_EVENT_SEND_COMPLETE. This means that the transfer failed when the value of the object that the pointer size points to, is 0xFFFFFFFF in the callback event USB_DEV_EVENT_SEND_COMPLETE. The application reserves the buffer until it receives a callback event indicating that the data is sent.

**Return Value**

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

### 4.2.4 USB_Class_HID_Cancel()

Cancels all the uncompleted transfers in the specified endpoint.

**Synopsis**

```
usb_status USB_Class_HID_Cancel
(
         hid_handle_t            handle,
          uint8_t                ep_num,
          uint8_t               direction
)
```

**Parameters**

*handle* *[in]* – HID class handler

*ep_num* *[in]* – endpoint number

direction *[in]* – direction of the endpoint

**Description**

The application calls this API function to cancel all the uncompleted transfers in the specified endpoint.

**Return Value**

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

### 4.2.5 USB_Class_HID_Recv_Data()

Sends the HID data.

**Synopsis**

```
usb_status USB_Class_HID_Recv_Data
(
              hid_handle_t            handle,
              uint8_t                 ep_num,
              uint8_t*                app_buff,
              uint32_t                size
)
```

**Parameters**

*handle* *[in]* – HID class handler

*ep_num* *[in]* – endpoint number

*app_buff* *[in]* – buffer to recv

*size* *[in]* – buffer length to recv

**Description**

The application calls this API function to send HID data specified by *app_buff* and *size*. Once the data is sent, the application layer receives a callback event USB_DEV_EVENT_DATA_RECEIVED. The transfer fails when the value of the object that the pointer size points is 0xFFFFFFFF in callback event USB_DEV_EVENT_DATA_RECEIVED. The application reserves the buffer until it receives a callback event indicating that the data is received.

**Return Value**

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

### 4.2.6 USB_Class_HID_Get_Speed()

Get the USB speed of the HID device.

**Synopsis**

```
usb_status USB_Class_HID_Get_Speed
(
    hid_handle_t          handle,
    uint16_t *            speed
)
```

**Parameters**

| | |
|---|---|
| `handle` | `[in]` – HID class handler |
| speed | `[out]` – current speed |

**Description**

The application calls this API function to get the USB speed of the HID device.

**Return Value**

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

## 4.3 MSC class API function listings

### 4.3.1 USB_Class_MSC_Init()

Initializes the MSC class.

**Synopsis**

```
usb_status USB_Class_MSC_Init
(
uint8_t              controller_id,
msc_config_struct_t*       msd_config_ptr,
msd_handle_t*        msd_handle
)
```

**Parameters**

| | |
|---|---|
| *`controller_id`* | *`[in]`* – controller ID, like USB_CONTROLLER_KHCI_0 |
| *`msd_config_ptr`* | *`[in]`* – MSD configuration structure, see `msc_config_struct_t` |

**Note**

In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

| | |
|---|---|
| *`msd_handle`* | *`[out]`* – pointer point to the initialized MSD class, see `msd_handle_t` |

**Description**

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

**Return Value**

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

### 4.3.2 USB_Class_MSC_Deinit()

Deinitializes the MSC class.

**Synopsis**

```
usb_status USB_Class_MSC_Deinit
(
        msd_handle_t           msd_handle
)
```

**Parameters**

*msd_handle* [in] – MSD class handler

**Description**

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

**Return Value**

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

### 4.3.3 USB_Class_MSC_Get_Speed()

Get the USB speed of the MSD device.

**Synopsis**

```
usb_status USB_Class_MSC_Get_Speed
(
    msd_handle_t         handle,
    uint16_t *           speed
)
```

**Parameters**

handle [in] – MSD class handler

speed [out] – current speed

**Description**

The application calls this API function to get the USB speed of the MSD device.

**Return Value**

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

## 4.4 Audio class API function listings

### 4.4.1 USB_Class_Audio_Init()

Initializes the AUDIO class.

**Synopsis**

```
usb_status USB_Class_Audio_Init
(
uint8_t                controller_id,
audio_config_struct_t*     audio_config_ptr,
audio_handle_t*            audioHandle
)
```

**Parameters**

*`controller_id`* *`[in]`* – controller ID, such as USB_CONTROLLER_KHCI_0

*`audio_config_ptr`* *`[in]`* – AUDIO configuration structure, see `audio_config_struct_t`

**Note**

> In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

`audioHandle` *`[out]`* – pointer point to the initialized AUDIO class, see `audio_handle_t`

**Description**

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

**Return Value**

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

### 4.4.2 USB_Class_Audio_Deinit()

Un-initializes the AUDIO class.

**Synopsis**

```
usb_status USB_Class_Audio_Deinit
(
         audio_handle_t          handle
)
```

**Parameters**

| | | |
|---|---|---|
| `handle` | `[in]` – | AUDIO class handler |

**Description**

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

**Return Value**

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

### 4.4.3 USB_Class_Audio_Recv_Data()

Receives the AUDIO data.

**Synopsis**

```
usb_status USB_Class_Audio_Recv_Data
(
         audio_handle_t          audio_handle,
         uint8_t                 ep_num,
         uint8_t*                app_buff,
         uint32_t                size
)
```

**Parameters**

| | | |
|---|---|---|
| *`audio_handle`* | *`[in]`* – | AUDIO class handler |
| *`ep_num`* | *`[in]`* – | endpoint number |
| *`app_buff`* | *`[in]`* – | buffer to save the data from the host |
| *`size`* | *`[in]`* – | buffer length to receive |

**Description**

The application calls this API function to receive data from host. Once the data is received, the application layer receives a callback event USB_DEV_EVENT_DATA_RECEIVED.

**Return Value**

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

### 4.4.4 USB_Class_Audio_Send_Data()

Sends the AUDIO data.

**Synopsis**

```
usb_status USB_Class_Audio_Send_Data
(
        audio_handle_t          handle,
         uint8_t                ep_num,
        uint8_t*                app_buff,
        uint32_t                size
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* – | AUDIO class handler |
| *ep_num* | *[in]* – | endpoint number |
| *app_buff* | *[in]* – | buffer to send |
| *size* | *[in]* – | buffer length to send |

**Description**

The application calls this API function to send data specified by *app_buff* and *size*. Once the data is sent, the application layer receives a callback event USB_DEV_EVENT_SEND_COMPLETE. The application reserves the buffer until it receives a callback event indicating that the data is sent.

**Return Value**

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

### 4.4.5 USB_Class_Audio_Cancel()

Cancels all the uncompleted transfers in the specified endpoint.

**Synopsis**

```
usb_status USB_Class_Audio_Cancel
(
        audio_handle_t          handle,
         uint8_t                ep_num,
         uint8_t               direction
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* – | AUDIO class handler |
| ep_num | [in] – | endpoint number |

direction *[in]* – direction of the endpoint

**Description**

The application calls this API function to cancel all the uncompleted transfers in the specified endpoint.

**Return Value**

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

### 4.4.6 USB_Class_Audio_Get_Speed()

Get the USB speed of the Audio device.

**Synopsis**

```
usb_status USB_Class_Audio_Get_Speed
(
    audio_handle_t       audio_handle,
    uint16_t *           speed
)
```

**Parameters**

audio_handle [in] – Audio class handler

speed [out] – current speed

**Description**

The application calls this API function to get the USB speed of the Audio device.

**Return Value**

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

## 4.5 PHDC class API function listings

### 4.5.1 USB_Class_PHDC_Init()

Initializes the PHDC class.

**Synopsis**

```
usb_status USB_Class_PHDC_Init
(
               uint8_t                 controller_id,
               phdc_config_struct_t*   phdc_config_ptr,
               phdc_handle_t*          phdcHandle
)
```

**Parameters**

| | | |
|---|---|---|
| *controller_id* | *[in]* | – controller ID, like USB_CONTROLLER_KHCI_0 |
| phdc_config_ptr | [in] | – PHDC configuration structure, see phdc_config_struct_t |

**Note**

> In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

| | | |
|---|---|---|
| phdcHandle | [out] | – pointer point to the initialized PHDC class, see phdc_handle_t |

**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)

## 4.5.2 USB_Class_PHDC_Deinit()

Un-initializes the PHDC class.

**Synopsis**

```
usb_status USB_Class_PHDC_Deinit
(
        phdc_handle_t           handle
)
```

**Parameters**

| | | |
|---|---|---|
| handle | [in] | – PHDC class handler |

**Description**

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

**Return Value**

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

## 4.5.3 USB_Class_PHDC_Recv_Data()

Receives the PHDC data.

**Synopsis**

```
usb_status USB_Class_PHDC_Recv_Data
```

```
(
              phdc_handle_t            handle,
              uint8_t                  qos,
              uint8_t*                 buff_ptr,
              int32_t                  size
)
```

**Parameters**

| | |
|---|---|
| handle | `[in]` – PHDC class handler |
| qos | `[in]` – the qos of the transfer |
| buff_ptr | `[in]` – buffer to save the data from the host |
| *size* | `[in]` – buffer length to receive |

**Description**

The application calls this API function to receive data from host. Once the data is received, the application layer receives a callback event USB_DEV_EVENT_DATA_RECEIVED. The transfer failed when the value of the object that the pointer size points is 0xFFFFFFFF in callback event USB_DEV_EVENT_DATA_RECEIVED.

**Return Value**

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

### 4.5.4 USB_Class_PHDC_Send_Data()

Sends the PHDC data.

**Synopsis**

```
usb_status USB_Class_PHDC_Send_Data
(
             phdc_handle_t             handle,
              bool                      meta_data,
              uint8_t                   num_tfr,
              uint8_t                  qos,
               uint8_t*                  app_buff ,
               uint32_t                  size
)
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – PHDC class handler |
| *meta_data* | *[in]* – the packet is metadata or not |
| *num_tfr* | *[in]* – the number of transfers |

| | | |
|---|---|---|
| *qos* | *[in]* – | current qos of the transfer |
| *app_buff* | *[in]* – | buffer to send |
| *size* | *[in]* – | buffer length to send |

**Description**

The application calls this API function to send data specified by `app_buff` and `size`. Once the data is sent, the application layer receives a callback event USB_DEV_EVENT_SEND_COMPLETE. The application reserves the buffer until it receives a callback event stating that the data is sent. The transfer failed when the value of the object that the pointer size points is 0xFFFFFFFF in callback event USB_DEV_EVENT_SEND_COMPLETE.

**Return Value**

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

### 4.5.5 USB_Class_PHDC_Cancel()

Cancels all the uncompleted transfers in the specified endpoint.

**Synopsis**

```
usb_status USB_Class_PHDC_Cancel
(
        phdc_handle_t          handle,
        uint8_t                ep_num,
        uint8_t                direction
)
```

**Parameters**

| | | |
|---|---|---|
| *handle* | *[in]* – | PHDC class handler |
| *ep_num* | *[in]* – | endpoint number |
| *direction* | *[in]* – | direction of the endpoint |

**Description**

The application calls this API function to cancel all the uncompleted transfers in the specified endpoint.

**Return Value**

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

### 4.5.6 USB_Class_PHDC_Get_Speed()

Get the USB speed of the PHDC device.

**Synopsis**

```
usb_status USB_Class_PHDC_Get_Speed
(
    phdc_handle_t        handle,
    uint16_t *           speed
)
```

**Parameters**

handle [in] – PHDC class handler

speed [out] – current speed

**Description**

The application calls this API function to get the USB speed of the PHDC device.

**Return Value**

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

## 4.6 Composite class API function listings

### 4.6.1 USB_Composite_Init()

Initializes the composite class.

**Synopsis**

```
usb_status USB_Composite_Init
(
uint8_t                        controller_id,
composite_config_struct_t*     composite_callback_ptr,
composite_handle_t*            compositeHandle
)
```

**Parameters**

*controller_id* *[in]* – controller ID, like USB_CONTROLLER_KHCI_0

composite_callback_ptr [in] – composite configuration structure, see composite_config_struct_t

compositeHandle [out] – pointer point to the initialized composite class, see composite_handle_t

**Note**

In the KSDK 1.3, board_init_callback is added to this configuration structure. Initialize it with an appropriate board initialization function.

**Description**

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

**Return Value**

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

## 4.6.2 USB_Composite_Deinit()

Un-initializes the Composite class.

**Synopsis**

```
usb_status USB_Composite_Deinit
(
        composite_handle_t           handle
)
```

**Parameters**

| | |
|---|---|
| `handle` | `[in]` – composite class handler |

**Description**

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

**Return Value**

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

## 4.6.3 USB_Composite_Get_Speed()

Get the USB speed of the Composite device.

**Synopsis**

```
usb_status USB_Composite_Get_Speed
(
    composite_handle_t   handle,
    uint16_t *           speed
)
```

**Parameters**

handle `[in]` – Composite class handler

speed `[out]` – current speed

**Description**

The application calls this API function to get the USB speed of the Composite device.

**Return Value**

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

# Chapter 5 USB Device Descriptor

During the enumeration process, the host needs to get the device’s descriptor. The USB Device Stack defines a set of callback functions to get the device’s descriptor from the application and then passes it to the host to finish the enumeration process.

## 5.1 get_desc

**Synopsis**

```
uint8_t (_CODE_PTR_ get_desc)
(
uint32_t             handle,
uint8_t        type,
uint8_t        desc_index,
uint16_t             index,
uint8_t * *          descriptor,
uint32_t *           size
);
```

**Parameters**

*`handle`* *`[in]`* – class handler returned by the class initialization API

*`type`* *`[in]`* – descriptor type

*`desc_index`* *`[in]`* – index of descriptor

*`index`* *`[in]`* – language ID if the type is USB_DESC_TYPE_STR

*`descriptor`* *`[out]`* – pointer point to the buffer to hold the descriptor

*`size`* *`[out]`* – descriptor size

**Description**

This callback function needs to be implemented by the application to provide different types of descriptor to the USB Device Stack, and the specified descriptor will be returned through the descriptor and size.

The following descriptors need to be supported in the implementation:

- USB_DESC_TYPE_DEV
- USB_DESC_TYPE_CFG
- USB_DESC_TYPE_STR
- USB_DESC_TYPE_DEV_QUALIFIER
- USB_DESC_TYPE_OTHER_SPEED_CFG

For different classes, the following descriptor may need to be supported:

- USB_HID_DESCRIPTOR
- USB_REPORT_DESCRIPTOR

**Return Value**

- **0** (success)
- **Others** (failure)

## 5.2 get_desc_interface

**Synopsis**

```
uint8_t (_CODE_PTR_ get_desc_interface)
(
uint32_t             handle,
uint8_t       interface,
uint8_t*             alt_interface
);
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – class handler returned by the class initialization API |
| *interface* | *[in]* – interface index |
| *alt_interface* | *[out]* – alternate setting for the interface |

**Description**

This callback function needs to be implemented by the application to get the current alternate setting for the specified interface.

**Return Value**

- **0** (success)
- **Others** (failure)

## 5.3 set_desc_interface

**Synopsis**

```
uint8_t (_CODE_PTR_ set_desc_interface)
(
uint32_t             handle,
uint8_t        interface,
uint8_t        alt_interface
);
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – class handler returned by the class initialization API |
| *interface* | *[in]* – interface index |
| *alt_interface* | *[in]* – alternate setting for the interface |

**Description**

This callback function needs to be implemented by the application to set the current alternate setting for the specified interface.

**Return Value**

- **0** (success)
- **Others** (failure)

## 5.4 set_configuration

**Synopsis**

```
uint8_t (_CODE_PTR_ set_configuration)
(
uint32_t             handle,
uint8_t        config
);
```

**Parameters**

| | |
|---|---|
| *handle* | *[in]* – The class handler returned by the class initialization API |
| *config* | *[in]* – The configuration index |

**Description**

This callback function needs to be implemented by the application to get to know which configuration is active by the host.

**Return Value**

- 0 (success)
- Others (failure)

## 5.5 get_desc_entity

### Synopsis

```
uint8_t (_CODE_PTR_ get_desc_entity)
(
uint32_t            handle,
entity_type         type,
uint32_t *          object
)
```

### Parameters

| | |
|---|---|
| *handle* | *[in]* – class handler returned by the class initialization API |
| *type* | *[in]* – entity type need to be obtained |
| *object* | *[out]* – target entity object pointer |

### Description

This callback function needs to be implemented by the application to provide descriptor/device related information to the USB Device stack.

The type could be:

- USB_CLASS_INFO

  Must be implemented by all kinds of devices to provide all the descriptor related information

- USB_AUDIO_UNITS

  Must be implemented by the AUDIO device to provide the AUDIO units related information

- USB_MSC_LBA_INFO

  Must be implemented by the MSC device to provide mass storage LAB related information

- USB_COMPOSITE_INFO

  Must be implemented by the composite device to provide composite device's descriptor related information.

- USB_RNDIS_INFO

  Must be implemented by the CDC RNDIS device to provide RNDIS related information

- USB_CLASS_INTERFACE_INDEX_INFO

  Must be implemented by the composite device to provide the class index

### Return Value

- **0** (success)
- **Others** (failure)

# Chapter 6 USB Device Configuration

The USB Device stack supports different configurations customized by the user in different cases.

The configuration varies depend on different boards, so you can get the configuration file in `<USB_ROOT>/usb_core/device/include/<SOC_NAME>`.

The following configuration items are supported in the USB Device Stack:

- USBCFG_DEV_KHCI
  - 1 indicates that the KHCI controller (Full-Speed) is enabled.
  - 0 indicates that the KHCI controller (Full-Speed) is disabled.
- USBCFG_DEV_EHCI
  - 1 indicates that the EHCI controller (High-Speed) is enabled.
  - 0 indicates that the EHCI controller (High-Speed) is disabled.
- USBCFG_DEV_KHCI_NUM
  - The MACRO indicates how many KHCI devices can be active at the same time. When this MACRO is bigger, more RAM is needed. If USBCFG_DEV_KHCI is nonzero, the default value is 1. Otherwise, the value is 0.
- USBCFG_DEV_EHCI_NUM
  - The MACRO indicates how many EHCI devices can be active at the same time. When this MACRO is bigger, more RAM is needed. If USBCFG_DEV_EHCI is nonzero, the default value is 1. Otherwise, the value is 0.
- USBCFG_DEV_NUM
  - The MACRO indicates how many devices can be active at the same time. When this MACRO is bigger, more RAM is needed. The default value is sum of USBCFG_DEV_KHCI_NUM and USBCFG_DEV_EHCI_NUM.
- USBCFG_DEV_HID
  - 1 indicates that the HID class driver is enabled.
  - 0 indicates that the HID class driver is disabled.
- USBCFG_DEV_PHDC
  - 1 indicates that the PHDC class driver is enabled.
  - 0 indicates that the PHDC class driver is disabled.
- USBCFG_DEV_AUDIO
  - 1 indicates that the AUDIO class driver is enabled.
  - 0 indicates that the AUDIO class driver is disabled.

- USBCFG_DEV_CDC
  - 1 indicates that the CDC class driver is enabled.
  - 0 indicates that the CDC class driver is disabled.
- USBCFG_DEV_MSC
  - 1 indicates that the MSC class driver is enabled.
  - 0 indicates that the MSC class driver is disabled.
- USBCFG_DEV_SELF_POWER
  - 1 indicates self power.
  - 0 indicates bus power.
- USBCFG_DEV_REMOTE_WAKEUP
  - 1 indicates that remote wakeup is enabled.
  - 0 indicates that remote wakeup is disabled.
- USBCFG_DEV_MAX_ENDPOINTS
  - The MACRO indicates how many endpoints can be used in a device. When this MACRO is bigger, more RAM is needed. The default value is 6.
- USBCFG_DEV_MAX_XDS
  - The MACRO indicates how many internal transfer descriptors can be used in a device. When this MACRO is bigger, more RAM is needed. The default value is 12.
- USBCFG_DEV_MAX_CLASS_OBJECT
  - The MACRO indicates how many instances can be supported for one class type device. When this MACRO is bigger, more RAM is needed. The default value is 1.
- USBCFG_KHCI_4BYTE_ALIGN_FIX
  - Effective when USBCFG_DEV_EHCI is nonzero.
  - The Full-Speed controller requires that all the buffers used for the transfer need to be 4 bytes aligned (both the start address and the length). If the application can guarantee it, then the MACRO USBCFG_KHCI_4BYTE_ALIGN_FIX can be set to 0. Otherwise, it needs to set to 1, and then the USB Device Stack will use an internal 4 bytes aligned buffer to replace the buffer provided by the user so that the above requirement can be meet. The internal buffer size is assigned by the MACRO.
- USBCFG_DEV_KHCI_SWAP_BUF_MAX.
  - Effective when USBCFG_DEV_EHCI is nonzero.
  - Effective when USBCFG_KHCI_4BYTE_ALIGN_FIX is nonzero.
- USBCFG_DEV_KHCI_ADVANCED_ERROR_HANDLING

    - 1 indicates that the error event will be sent to the application.
    - 0 indicates that the error event will not be sent to the application.
    - Effective for the Full-Speed controller only.
- USBCFG_DEV_EHCI_MAX_ENDPOINTS
    - The MACRO indicates how many endpoints can be used in a high-speed device. The default value is depended on the Chip.
    - Effective for the High-Speed controller only.
- USBCFG_DEV_EHCI_MAX_DTD
    - The MACRO indicates how many DTD can be used in a high-speed device. When this MACRO is bigger, more RAM is needed. The default value is 16.
    - Effective for the High-Speed controller only.
- USBCFG_DEV_EHCI_ADVANCED_ERROR_HANDLING
    - 1 indicates that the error event will be sent to the application.
    - 0 indicates that the error event will not be sent to the application.
    - Effective for the High-Speed controller only.
- USBCFG_DEV_KEEP_ALIVE_MODE
    - 1 indicates that keep alive is enabled.
    - 0 indicates that keep alive disabled.
    - Effective for the Full-Speed controller only.
- USBCFG_DEV_BUFF_PROPERTY_CACHEABLE
    - 1 indicates that the cache maintenance in USB Device Stack is enabled.
    - 0 indicates that the cache maintenance in USB Device Stack is disabled.
    - If the application does not use the cacheable memory in the USB transfer, this MACRO needs to be set to 0. Otherwise, it needs to be set to 1, and the application needs to make sure that the cacheable memory is CACHE LINE size aligned (both start address and length).
- USBCFG_DEV_ADVANCED_SUSPEND_RESUME
    - Whether the suspend/resume needs to be enabled. It can be set to 0 only, because suspend/resume is not implemented yet.
- USBCFG_DEV_ADVANCED_CANCEL_ENABLE
    - Whether the cancel operation needs to be enabled. The default value is 1.
- USBCFG_DEV_DETACH_ENABLE

- Whether the device detach function needs to be enabled. The default value is 0.

- USBCFG_DEV_IO_DETACH_ENABLE
  - Whether the device detach function uses IO to implement when device detach function is enabled. The default value is 0.

# Chapter 7 USB Device Data structures

This section describes the data structures that need to be used from the application.

## 7.1 USB common controller driver

### 7.1.1 usb_ep_struct_t

**Description**

Obtains the endpoint data structure.

**Synopsis**

```
typedef struct _usb_ep_struct
{
uint8_t        ep_num;
uint8_t        type;
uint8_t        direction;
uint32_t                  size;
} usb_ep_struct_t;
```

**Fields**

`ep_number` – endpoint number

`type` – type of endpoint

`direction` – direction of endpoint

`size` – maximum packet size of endpoint

### 7.1.2 usb_endpoints_t

**Description**

Obtains the endpoint group.

**Synopsis**

```
typedef struct _usb_endpoints
{
    uint8_t                count;
    usb_ep_struct_t*       ep;
} usb_endpoints_t;
```

**Fields**

`count` – how many endpoints are described

`ep` – detailed information of each endpoint

### 7.1.3 usb_if_struct_t

**Description**

Obtains the interface data structure.

**Synopsis**

```
typedef struct _usb_if_struct
{
    uint8_t                 index;
    usb_endpoints_t  endpoints;
} usb_if_struct_t;
```

**Fields**

`index` – interface index

`endpoints` – endpoints in this interface

### 7.1.4 usb_interfaces_struct_t

**Description**

Obtains the interface group.

**Synopsis**

```
typedef struct _usb_interfaces_struct
{
    uint8_t                count;
    usb_if_struct_t*      interface;
} usb_interfaces_struct_t;
```

**Fields**

`count` – how many interfaces are described

`interface` – detailed information of each interface

### 7.1.5 usb_class_struct_t

**Description**

Obtains the class data structure.

**Synopsis**

```
typedef struct _usb_class_struct
{
    class_type                         type;
    usb_interfaces_struct_t      interfaces;
} usb_class_struct_t;
```

**Fields**

*index* – class type

*interfaces* – interfaces in this class

### 7.1.6 usb_composite_info_struct_t

**Description**

Obtains the composite information data structure.

**Synopsis**

```
typedef struct _usb_composite_info_struct
{
    uint8_t                     count;
    usb_class_struct_t*    class;
} usb_composite_info_struct_t;
```

**Fields**

count – how many classes in the composite device

class – detailed information of each class

### 7.1.7 usb_desc_request_notify_struct_t

**Description**

Obtains the data structure of the descriptor related callback function. The application needs to implement them and passes it as the configuration parameter.

**Synopsis**

```
typedef struct _usb_desc_request_notify_struct
{
#ifdef USBCFG_OTG
        uint32_t handle;
#endif
    uint8_t (_CODE_PTR_ get_desc)(uint32_t handle,uint8_t type,uint8_t desc_index,
        uint16_t index,uint8_t * *descriptor,uint32_t *size);
    uint8_t (_CODE_PTR_ get_desc_interface)(uint32_t handle,uint8_t interface,
        uint8_t * alt_interface);
    uint8_t (_CODE_PTR_ set_desc_interface)(uint32_t handle,uint8_t interface,
        uint8_t alt_interface);
    uint8_t (_CODE_PTR_ set_configuration)(uint32_t handle, uint8_t config);
    uint8_t (_CODE_PTR_ get_desc_entity)(uint32_t handle, entity_type type, uint32_t *
object);
} usb_desc_request_notify_struct_t;
```

**Fields**

`get_desc` – to get the descriptor whose type is specified by the type

`get_desc_interface` – to get the interface’s alternate setting

`set_desc_interface` – to set the interface’s alternate setting

`set_configuration` – to inform the application whose configuration is active

`get_desc_entity` – to get the descriptor/device related information

## 7.2 CDC class data structure

### 7.2.1 cdc_handle_t

**Description**

Represents the CDC class handle.

**Synopsis**

```
typedef uint32_t cdc_handle_t;
```

### 7.2.2 _ip_address

**Description**

Represents the IP address.

**Synopsis**

```
typedef uint32_t _ip_address;
```

### 7.2.3 cdc_app_data_struct_t

**Description**

Holds the information of CDC data struct.

**Synopsis**

```
typedef struct _cdc_app_data_struct
{
    uint8_t*                                  data_ptr;
    uint32_t                                  data_size;
}cdc_app_data_struct_t;
```

**Fields**

`data_ptr` - pointer to buffer

`data_size` - buffer size

### 7.2.4 usb_rndis_info_struct_t

**Description**

Holds the detailed information about the RNDIS.

**Synopsis**

```
typedef struct _usb_rndis_info_struct
{
    enet_address_t mac_address;
    _ip_address   ip_address;
    uint32_t      rndis_max_frame_size;
} usb_rndis_info_struct_t;
```

**Fields**

*mac_address* – MAC address

*ip_address* – IP address

*rndis_max_frame_size* – maximum frame size

### 7.2.5 cdc_config_struct_t

**Description**

Holds the detailed information about the CDC configuration.

**Synopsis**

```
typedef struct _cdc_config_struct
{
   usb_application_callback_struct_t        cdc_application_callback;
   usb_vendor_req_callback_struct_t         vendor_req_callback;
   usb_class_specific_callback_struct_t     class_specific_callback;
   usb_desc_request_notify_struct_t*        desc_callback_ptr;
  usb_board_init_callback_struct_t         board_init_callback;
}cdc_config_struct_t;
```

**Fields**

*cdc_application_callback* – application callback function to handle the Device status related event

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related events

board_init_callback - application callback function to handle board init

*desc_callback_ptr* – descriptor related callback function data structure

## 7.3 HID class data structure

### 7.3.1 hid_handle_t

**Description**

Represents the HID class handle.

**Synopsis**

```
typedef uint32_t hid_handle_t;
```

### 7.3.2 hid_config_struct_t

**Description**

Holds the detailed information about the HID class configuration.

**Synopsis**

```
typedef struct _hid_config_struct
{
   usb_application_callback_struct_t        hid_application_callback;
   usb_vendor_req_callback_struct_t         vendor_req_callback;
   usb_class_specific_callback_struct_t     class_specific_callback;
   usb_desc_request_notify_struct_t*        desc_callback_ptr;
  usb_board_init_callback_struct_t         board_init_callback;
}hid_config_struct_t;
```

**Fields**

*hid_application_callback* – application callback function to handle the Device status related event

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related event

board_init_callback - application callback function to handle board init

*desc_callback_ptr* – descriptor related callback function data structure.

## 7.4 MSC class data structure

### 7.4.1 msc_handle_t

**Description**

Represents the MSC class handle.

**Synopsis**

```
typedef uint32_t msc_handle_t;
```

### 7.4.2 msc_app_data_struct_t

**Description**

Holds the information of MSC app data struct.

**Synopsis**

```
typedef struct _msc_app_data_struct
{
    uint8_t*                                  data_ptr;
    uint32_t                                  data_size;
}msc_app_data_struct_t;
```

**Fields**

`data_ptr` - pointer to buffer

`data_size` - buffer size

### 7.4.3 lba_app_struct_t

**Description**

Holds the information used to perform the logical block access.

**Synopsis**

```
typedef struct _lba_app_struct
{
    uint32_t            offset;
    uint32_t            size;
    uint8_t*            buff_ptr;
}lba_app_struct_t;
```

**Fields**

*`offset`* – offset of target block need to access

*`size`* – size need to access

*`buff_ptr`* – used to save the content by access the target block

### 7.4.4 device_lba_info_struct_t

**Description**

Holds the detailed information about the LAB.

**Synopsis**

```
typedef struct _device_lba_info_struct
{
    uint32_t                total_lba_device_supports;
    uint32_t                length_of_each_lab_of_device;
    uint8_t                 num_lun_supported;
}device_lba_info_struct_t;
```

**Fields**

*total_lba_device_supports* – blocks number

*length_of_each_lab_of_device* – size of each block

*num_lun_supported* – number of LUN

### 7.4.5 msc_config_struct_t

**Description**

Holds the detailed information about the MSC configuration.

**Synopsis**

```
typedef struct _msc_config_struct
{
   usb_application_callback_struct_t        msc_application_callback;
   usb_vendor_req_callback_struct_t         vendor_req_callback;
   usb_class_specific_callback_struct_t     class_specific_callback;
   usb_desc_request_notify_struct_t*        desc_callback_ptr;
  usb_board_init_callback_struct_t         board_init_callback;
}msc_config_struct_t;
```

**Fields**

*msc_application_callback* – application callback function to handle the Device status related event

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related event

board_init_callback - application callback function to handle board init

*desc_callback_ptr* – descriptor related callback function data structure.

## 7.5 Audio class data structure

### 7.5.1 audio_handle_t

**Description**

Represents the AUDIO class handle.

**Synopsis**

```
typedef uint32_t audio_handle_t;
```

### 7.5.2 audio_app_data_struct_t

**Description**

Holds the information of audio app data struct.

**Synopsis**

```
typedef struct _audio_app_data_struct
{
    uint8_t*                                  data_ptr;
    uint32_t                                  data_size;
}audio_app_data_struct_t;
```

**Fields**

*data_ptr* - pointer to buffer

*data_size* - buffer size

### 7.5.3 audio_config_struct_t

**Description**

Holds the detailed information about the AUDIO configuration.

**Synopsis**

```
typedef struct _audio_config_struct
{
   usb_application_callback_struct_t        audio_application_callback;
   usb_vendor_req_callback_struct_t         vendor_req_callback;
   usb_class_specific_callback_struct_t     class_specific_callback;
  usb_board_init_callback_struct_t         board_init_callback;
   usb_desc_request_notify_struct_t*        desc_callback_ptr;
}audio_config_struct_t;
```

**Fields**

*audio_application_callback* – application callback function to handle the Device status related event

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related event

*board_init_callback* - application callback function to handle board

*initdesc_callback_ptr* – descriptor related callback function data structure.

## 7.6 PHDC class data structure

### 7.6.1 phdc_handle_t

**Description**

Represents the PHDC class handle.

**Synopsis**

```
typedef uint32_t phdc_handle_t;
```

### 7.6.2 phdc_app_data_struct_t

**Description**

Holds the buffer information along with the send complete event.

**Synopsis**

```
typedef struct _phdc_app_data_struct
{
    uint8_t   qos;
    uint8_t*  buffer_ptr;
    uint32_t  size;
} phdc_app_data_struct_t;
```

**Fields**

*qos* – the qos of the transfer

*buffer_ptr* – the buffer point

*size* – the buffer size

### 7.6.3 phdc_config_struct_t

**Description**

Holds the detailed information about the PHDC configuration.

**Synopsis**

```
typedef struct _phdc_config_struct
{
   usb_application_callback_struct_t        phdc_application_callback;
   usb_vendor_req_callback_struct_t         vendor_req_callback;
   usb_class_specific_callback_struct_t     class_specific_callback;
  usb_board_init_callback_struct_t         board_init_callback;
   usb_desc_request_notify_struct_t*        desc_callback_ptr;
}phdc_config_struct_t;
```

**Fields**

*phdc_application_callback* – application callback function to handle the Device status related event

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related event

board_init_callback - application callback function to handle board init

*desc_callback_ptr* – descriptor related callback function data structure.

## 7.7 Composite class data structure

### 7.7.1 composite_handle_t

**Description**

Represents the composite class handle.

**Synopsis**

```
typedef uint32_t composite_handle_t;
```

### 7.7.2 class_config_struct_t

**Description**

Holds the information one type of class configuration.

**Synopsis**

```
typedef struct _class_config_struct
 {
     usb_application_callback_struct_t          application_callback;
     usb_vendor_req_callback_struct_t           vendor_req_callback;
     usb_class_specific_callback_struct_t       class_specific_callback;
     usb_desc_request_notify_struct_t*          desc_callback_ptr;
    usb_board_init_callback_struct_t         board_init_callback;
     uint32_t                                   class_handle;
     class_type                                 type;
 }class_config_struct_t;
```

**Fields**

*application_callback* – application callback function to handle the Device status related event for the specified type of class

*vendor_req_callback* – application callback function to handle the vendor request related event, reserved for future use

*class_specific_callback* – application callback function to handle all the class related event for the specified type of class

*desc_callback_ptr* – descriptor related callback function data structure for the specified type of class.

*class_handle* – the handle of the class.

*board_init_callback* - application callback function to handle board init

*type* – class type

### 7.7.3 composite_config_struct_t

**Description**

Holds the detailed information about the MSC configuration.

**Synopsis**

```
typedef struct _composite_config_struct
{
     uint8_t                     count;
     class_config_struct_ptr           class_app_callback;
}composite_config_struct_t;
```

**Fields**

*`count`* – how many classes are supported for the composite device

*`class_app_callback`* – detailed information for each class

# Appendix

## A. USB device stack tasks

There are one task as follow:

| Task name | Priority macro | Priority value |
|---|---|---|
| DEV task | USB_DEVICE_TASK_PRIORITY | 6 |

**Note**

For RTOS, application unblocked tasks' priorities should be lower than the above priority.

# Chapter 8 Revision history

This table summarizes revisions to this document since the release of the previous version

| Revision History | | |
|---|---|---|
| **Revision number** | **Date** | **Substantive changes** |
| 1 | 04/2015 | KSDK 1.2.0 Release |
| 2 | 09/2015 | Updated Section 2.1 |

**How to Reach Us:**

**Home Page:**
www.freescale.com

**Web Support:**
www.freescale.com/support

Document Number: USBSDRM
Rev. 2
09/2015