Freescale USB Stack Device
API Reference Manual

Document Number: USBAPIRM
Rev. 10
05/2012

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: USBAPIRM
Rev. 10
05/2012

Revision history

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

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

Revision Number	Revision Date	Description of Changes
Rev. 1	05/2009	Alpha Customer Release.
Rev. 2	06/2009	Added support for ColdFire V1.
Rev. 3	09/2009	Launch Release. Customized for Medical Applications.
Rev. 4	10/2009	Added Mass Storage Class (MSC) API functions.
Rev. 5	06/2010	Rebranded Medical Applications USB Stack to Freescale USB Stack with PHDC.
Rev. 6	09/2010	Added Audio Device Class API functions and data structures
Rev. 7	12/2010	Added DFU Class API functions and data structures
Rev. 8	07/2011	Added Battery Charging API functions and data structures
Rev. 9	03/2012	Replaced the term "Freescale USB Stack with PHDC" with "Freescale USB Stack"
Rev. 10	05/2012	Added USB Video Device API functions and data structures

Chapter 1

Before Beginning

Chapter 2

USB Device API 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 Beginning

1.1 About this book

This book describes the Freescale USB Stack device and class API functions. It describes in detail the API functions that can be used to program the USB controller at various levels. Table 1-1 shows the summary of chapters included in this book.

Table 1-1. USBAPIRM summary

Chapter Title	Description
Before Beginning	This chapter provides the prerequisites of reading this book.
USB Device API Overview	This chapter gives an overview of the API functions and how to use them for developing new class and applications.
USB Device Layer API	This chapter discusses the USB device layer API functions.
USB Device Class API	This chapter discusses the USB device class layer API functions of the various classes provided in the software suite.
Data Structures	This chapter discusses the various data structures used in the USB device class layer API functions.
Reference Data Types	This chapter discusses the data types used to write USB device and class API functions.

1.2 Reference material

Use this book in conjunction with:

- *Freescale USB Stack Device Users Guide* (document USBUG)
- S08 USB Device Source Code
- ColdFire V1 USB Device Source Code
- ColdFire V2 USB Device Source Code
- Kinetis USB Device Source Code

For better understanding, refer to the following documents:

- USB Specification Revision 1.1
- USB Specification Revision 2.0
- S08 Core Reference
- ColdFire V1 Core Reference
- ColdFire V2 Core Reference
- Kinetis (ARM Cortex-M4) Core Reference

- CodeWarrior Help
- Battery Charging Specification Rev. 1.1

1.3 Acronyms and abbreviations

API	Application Programming Interface
CDC	Communication Device Class
CFV1	ColdFire V1 (MCF51JM128 CFV1 device is used in this document)
CFV2	ColdFire V2 (MCF52221, MCF52259, and MCF52277 CFV2 device is used in this document)
CDP	Charging Downstream Port
DCP	Dedicated Charging Port
DFU	Device Firmware Upgrade
HID	Human Interface Device
IDE	Integrated Development Environment
JM128	MCFJM128Device
MSD	Mass Storage Device
MSC	Mass Storage Class
PD	Portable Device
PHD	Personal Healthcare Device
PHDC	Personal Healthcare Device Class
QOS	Quality of Service
SDP	Standard Downstream Port
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 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
USB Device API Overview

2.1 Introduction

The Freescale USB Stack device API consists of the functions that can be used at the device level and the class level. This enables you to implement new classes. This document describes four generic class implementations: Communication Device Class (CDC), Human Interface Device (HID), Mass Storage Class (MSD), and Personal Healthcare Device Class (PHDC) API functions that are provided as part of the software suite. The API functions defined for these classes can be used to make applications.

The USB device software consists of the:

- Class Layer API
- Device Layer API

For better understanding, see the *Freescale USB Stack Stack Users Guide* (document USBUG).

2.2 USB Device

Figure 2-1 shows the USB device layers.

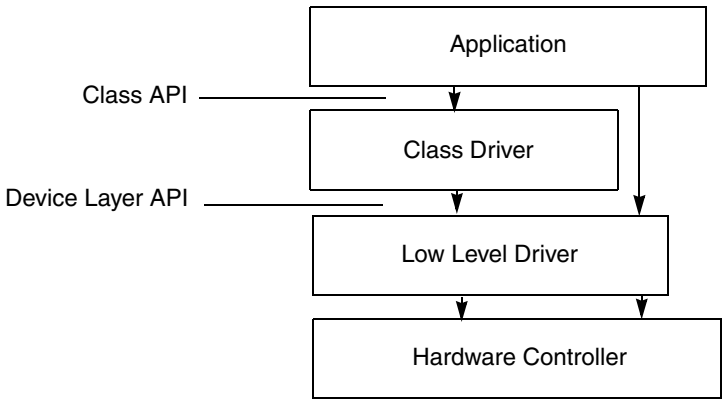

Figure 2-1. USB Device layers

The application usually sits on top of the stack. Based on the implementation, the application interfaces with the class driver, the low level driver, or sometimes with the controller directly.

The Freescale USB Stack software suite provides the flexibility to use the class API interface or the device layer API interface. However, both must not be used at the same time because this may lead to unwanted results.

2.3 API overview

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

Table 2-1 summarizes the device layer API functions.

Table 2-1. Summary of device layer API functions

No.	API Function	Description
1	USB_Device_Assert_Resume()	Resumes signal on the bus for remote wakeup
2	USB_Device_Cancel_Transfer()	Cancels a pending send or receive call
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_DeInit()	De-initializes a USB device controller
8	USB_Device_Init_EndPoint()	Initializes the endpoint provided as parameter to the API
9	USB_Device_Read_Setup_Data()	Reads the setup data for an endpoint
10	USB_Device_Recv_Data()	Copy the data received on an endpoint and sets the endpoint to receive the next set of data
11	USB_Device_Register_Service()	Registers the callback service for a type of event or endpoint
12	USB_Device_Send_Data()	Sends data on an endpoint
13	USB_Device_Set_Address()	Sets the address of a USB device controller
14	USB_Device_Set_Status()	Sets the internal USB device state
15	USB_Device_Shutdown()	Shuts down a USB device controller
16	USB_Device_Stall_EndPoint()	Stalls an endpoint in the specified direction
17	USB_Device_Unstall_EndPoint()	Unstalls a previously stalled endpoint
18	USB_Device_Unregister_Service()	Unregisters the callback service for a type of event or endpoint
19	USB_Device_Unstall_EndPoint()	Unstalls a previously stalled endpoint

Table 2-2 summarizes the CDC class API functions.

Table 2-2. Summary of CDC Class API functions

No.	API Function	Description
1	USB_Class_CDC_Init()	Initializes the CDC class
2	USB_Class_CDC_DeInit()	De-initializes the CDC class
3	USB_Class_CDC_Interface_CIC_Send_Data()	Sends the communication Interface information to the host
4	USB_Class_CDC_Interface_DIC_Recv_Data()	Receives the data from the host
5	USB_Class_CDC_Interface_DIC_Send_Data()	Sends the data to the host
6	USB_Class_CDC_Periodic_Task()	Periodic call to the class driver to complete pending tasks

Table 2-3 summarizes the HID class API functions.

Table 2-3. Summary of HID Class API functions

No.	API Function	Description
1	USB_Class_HID_Init()	Initializes the HID class
2	USB_Class_HID_DeInit()	De-initializes the HID class
3	USB_Class_HID_Periodic_Task()	Periodic call to the class driver to complete pending tasks
4	USB_Class_HID_Send_Data()	Sends the HID report to the host

Table 2-4 summarizes the MSC class API functions.

Table 2-4. Summary of MSC Class API functions

No.	API Function	Description
1	USB_Class_MSC_Init	Initializes the MSC class
2	USB_Class_MSC_DeInit	De-initializes the MSC class
2	USB_Class_MSC_Periodic_Task	Periodic call to the class driver to complete pending tasks

Table 2-5 summarizes the PHDC class API functions.

Table 2-5. Summary of PHDC Class API functions

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

Table 2-6 summarizes the Audio Class API functions.

Table 2-6. Summary of Audio Class API functions

No.	API Function	Description
1	USB_Class_Audio_Init()	Initializes the audio class
2	USB_Class_Audio_DeInit()	De-initializes the audio class
3	USB_Audio_Class_Send_Data()	Sends the audio data to the host
4	USB_Audio_Class_Recv_Data()	Receives the audio data from host

Table 2-7 summarizes the DFU class API functions

Table 2-7. Summary of DFU Class API functions

No	API Function	Description
1	USB_Class_DFU_Init	Initializes the DFU Class
2	USB_Class_DFU_DeInit()	De-initializes the DFU class

Table 2-8 summarizes the descriptor module API functions required by the class layers for application implementation. See Chapter 5“USB Descriptor API for more details on sample implementation of each API function.

Table 2-8. 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 wakeup 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 API

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

2.4.1 Using the Device Layer API

This section describes a sequence 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 3.1.6“_usb_device_init()” to initialize the low level driver and the controller.
2. Call 3.1.11“_usb_device_register_service()” to register service callback functions for the following:
   - USB_SERVICE_BUS_RESET
   - USB_SERVICE_SUSPEND
   - USB_SERVICE_SOF
   - USB_SERVICE_RESUME
   - USB_SERVICE_SLEEP

- USB_SERVICE_ERROR
- USB_SERVICE_STALL

3. Call 3.1.11“_usb_device_register_service()” to register service call back functions for control and non-control endpoints.
4. Call 3.1.8“_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 De-initialization flow

To de-initialize the driver layer, the class layer must:

1. Call 3.1.7“_usb_device_deinit()” to de-initialize low level driver and the controller.
2. Call 3.1.17“_usb_device_unregister_service()” to unregister service callback functions for the following:
   - USB_SERVICE_BUS_RESET
   - USB_SERVICE_SUSPEND
   - USB_SERVICE_SOF
   - USB_SERVICE_RESUME
   - USB_SERVICE_SLEEP
   - USB_SERVICE_ERROR
   - USB_SERVICE_STALL
3. Call 3.1.17“_usb_device_unregister_service()” to unregister callback functions for control and non-control endpoints.

2.4.1.3 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 3.1.12“_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.

2.4.1.4 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 3.1.11“_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 3.1.10“_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. In the case where 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 4.1.1“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 4.1.3“USB_Class_CDC_Interface_CIC_Send_Data()” and 4.1.5“USB_Class_CDC_Interface_DIC_Send_Data()” to send data to the host through the device layers, when required.
4. 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 4.2.1“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 4.2.4“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 4.3.1“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 storage device before sending it on USB bus. It reads data from mass storage device to driver buffer.
4. Callback function is called with the USB_MSC_DEVICE_WRITE_REQUEST event to copy data from USB driver buffer to Storage device. It reads data from driver buffer to mass storage device.

2.4.5 PHDC Class Layer API

To use PHDC class layer API functions from the application:

1. Call 4.4.1“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 4.4.4“USB_Class_PHDC_Send_Data()” to send data to the host through the device layers, when required.
4. Callback function is called with the USB_APP_DATA_RECEIVED event that implies reception of data from the host.

2.4.6 Audio Class Layer API

To use Audio class layer API functions from the application:

1. Call 4.5.1“USB_Class_Audio_Init()" function to initialize the class driver, all the layers below it, and the device controller. The event callback function is also passed as parameter to the Init function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application can consider the USB enumeration complete and can move into the connected state where it can exchange audio data with the Host.
3. Call 4.5.3“USB_Class_Audio_Send_Data()” to send data to the host through the device layers, when required.
4. Callback function is called with the USB_APP_DATA_SEND_COMPLETED event that implies sending of data to the host.
5. Call 4.5.4“USB_Audio_Class_Recv_Data()” to get data from the host through the device layers, when required.
6. Callback function is called with the USB_APP_DATA_RECEIVED event that implies receiving of Audio data from the host.

2.4.7 DFU Class Layer API

To use DFU class layer API functions from the application:

1. Call 4.6.1“USB_Class_DFU_Init()” function to initialize the class driver, all the layers below it, and the device controller. The event callback function is also passed as parameter to the Init function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application can consider the USB enumeration complete and can move into the connected state where it can exchange DFU firmware with the Host.

2.4.8 USB Battery Charging

The USB Battery Charging driver interacts with the top application layer in order to start the activities like: VBUS detection, data pin contact detection, port type detection or to inform about the operation status through the registered application callback.

On the other side the driver shall control the hardware support for charging detection purpose, in order to start the appropriate either voltage or current sources, current sink circuit or voltage comparators, according to the current step.

The Battery Charging Detection Driver shall control also the D+ pull-up resistor (to detect the type of the charging port at which is it attached) through the USB controller driver.

A typical relationship between the Battery Charger driver and the other components (application layer, hardware support, USB controller driver) is depicted in the Figure 2-2.

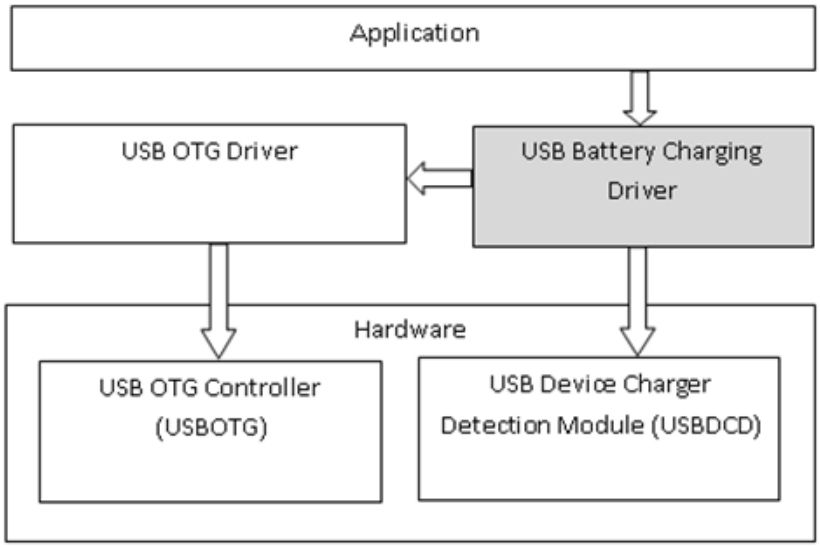

Figure 2-2. The USB battery charging software architecture

2.4.8.1

Table 2-9 describes the list of Battery charging functions and their uses:

Table 2-9. Summary of battery charging functions

No.	API Function	Description
1	_usb_batt_chg_init()	Initialize the Battery Charging variable structure and hardware support.
2	_usb_batt_chg_uninit()	De-allocates the Battery Charging driver's static variables and reset the HW support.
3	_usb_batt_chg_register_callback()	Register the function callback used by the application for any event received from the Battery Charging software support.
4	_usb_batt_chg_task()	This function is responsible to monitor any change in the USB battery charging mechanism.
5	_usb_batt_chg_ext_isr()	Function to enable the pending external interrupt from the VBUS detection IC.

2.4.8.2 Battery Charging API

The following steps explain how to write a Batter Charging application:

1. Write the functions to be passed to the Battery Charging driver through the Initialization
2. Declare a Battery Charging initialization structure then initialize it.
3. Write the Battery Charging callback function. This function will manage the Battery Charging events.
4. Write the function to initialize the pin used by the external IC to inform a status change on the VBUS voltage.
5. Declares the interrupt function for the VBUS status change and place the *_usb_batt_chg_ext_isr()* function into it.
6. Place a call to the *_usb_batt_chg_init()* function and a call to the *_usb_batt_chg_register_callback()* function into the initialization part of the application.
7. Into the test application task function.

2.4.9 USB Video Device API

2.4.9.1 Introduction

The video device class applies to all devices or functions embedded in composite devices used to manipulate video and video-related functionality. This includes devices such as desktop video cameras (or "webcams"), Digital video cameras/desks (digital camcorders) and so on.

2.4.9.2 USB Video Device

The Video class provides an Video control interface and an Video stream interface. While the Video control interface controls the functional behavior of particular Video function, Video stream interface is used to interchange Video data streams between the host and the Video function.

2.4.9.3 API overview

Table 2-10 summarizes the Video class API functions:

Table 2-10. Summary of Video Class API Functions

No.	API Function	Description
1	USB_Class_Video_Init()	Initializes the video class
2	USB_Class_Video_Send_Data()	Sends the video data to the host

2.4.9.4 Using API

To use Video class layer API functions from the application:

1. Call “USB_Class_Video_Init()” function to initialize the class driver, all the layers below it, and the device controller. The event callback function is also passed as parameter to the Init function.
2. When the callback function is called with the USB_APP_ENUM_COMPLETE event, the application can consider the USB enumeration complete and can move into the connected state where it can exchange Video data with the Host.
3. Call “USB_Class_Video_Send_Data()” to send data to the host through the device layers, when required.
4. Callback function is called with the USB_APP_DATA_SEND_COMPLETE event that implies sending of data to the host.

Chapter 3
USB Device Layer API

This section discusses the device layer API functions in detail.

3.1 USB Device Layer API function listings

3.1.1 _usb_device_assert_resume()

Resumes the USB host. Available for USB 2.0 Device API only.

Synopsis

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

Parameters

handle [in] — USB Device handle

Description

The function signals the host to start the resume process. This function is called when the device needs to send the data to the USB host and the USB bus is in suspend state.

Return Value

None

3.1.2 _usb_device_cancel_transfer()

Cancels the transfer on the endpoint.

Synopsis

```
uint_8 _usb_device_cancel_transfer(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uint_8 direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — USB endpoint number for the transfer

direction [in] — Direction of the buffer; 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 does not change the status of the endpoint to idle. If the transfer is active, the function calls this device layer API function to terminate all the transfers queued on the endpoint and sets the status to idle. This function blocks until the transfer cancellation is complete.

Return Value

- **USB_OK** (success)
- **USBERR_UNKNOWN_ERROR** (unknowm error)
- **USBERR_NOT_SUPPORTED** (failure: queuing of requests not supported)

See Also:

3.1.12 "_usb_device_send_data()"

3.1.10 "_usb_device_recv_data()"

3.1.5 "_usb_device_get_transfer_status()"

3.1.3 _usb_device_deinit_endpoint()

Disables the endpoint for the USB device controller.

Synopsis

```
uint_8 _usb_device_deinit_endpoint(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uint_8 direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — USB endpoint number

direction [in] — Direction of transfer

Description

The function disables the specified endpoint.

Return Value

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

See Also:

3.1.6 "_usb_device_init()"

3.1.8 "_usb_device_init_endpoint()"

3.1.4 _usb_device_get_status()

Gets the internal USB device state.

Synopsis

```
uint_8 _usb_device_get_status(
        _usb_device_handle handle,
        uint_8 component,
        uint_16_ptr status);
```

Parameters

handle [in] — USB Device handle

component [in] — Components defined for *USB Chapter 9* Get_Status/Set_Status calls, the possible values are:

USB_STATUS_ADDRESS
USB_STATUS_CURRENT_CONFIG
USB_STATUS_DEVICE
USB_STATUS_DEVICE_STATE
USB_STATUS_INTERFACE
USB_STATUS_SOF_COUNT
USB_STATUS_TEST_MODE
USB_STATUS_ENDPOINT —The LSB nibble carries the endpoint number

status [out] — Current status of the component

Description

The function calls this device layer API function to retrieve the current status for the defined component used for *USB Chapter 9* Get_Status or Set_Status USB framework calls. This call is usually made when the Get_Status USB framework call is received from the USB host.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: status of last transfer is unknown)

See Also:

3.1.14 "_usb_device_set_status()"

3.1.5 _usb_device_get_transfer_status()

Gets the status of the last transfer on the endpoint.

Synopsis

```
uint_8 _usb_device_get_transfer_status(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uint_8 direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — USB endpoint number

direction [in] — Direction of the buffer; 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, the function may call this device layer API function to check the status of that transfer.

Return Value

- **USB_STATUS_DISABLED** — Endpoint is disabled
- **USB_STATUS_IDLE** — No activity at the endpoint
- **USB_STATUS_STALLED** — Endpoint is stalled
- **USB_STATUS_TRANSFER_IN_PROGRESS** — Data is being transferred or received

See Also:

3.1.2 "_usb_device_cancel_transfer()"

3.1.6 _usb_device_init()

Initializes the low level device driver and the USB device controller.

Synopsis

```
uint_8 _usb_device_init (
        uint_8 device_number,
        _usb_device_handle _PTR_ handle,
        uint_8 number_of_endpoints)
```

Parameters

device_number [in] — USB device controller to initialize

handle [out] — Pointer to a USB Device handle

number_of_endpoints [in] — Maximum number of endpoints to be supported by the device

Description

The function does the following:

- Initializes the USB device-specific data structures
- Calls the device-specific initialization function

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_NUM_OF_ENDPOINTS** (failure: reported when the number of endpoints are invalid for initialization)

See Also:

3.1.15 "_usb_device_shutdown()"

3.1.7 _usb_device_deinit()

De-initializes low level device driver and the USB device controller.

Synopsis

```
uint_8 _usb_device_deinit(void)
```

Parameters

None

Description

The function does the following:

- De-initializes the USB device-specific data structures
- Call the device-specific de-initialize function

Return value

USB_OK (success)

See Also:

3.1.15 “_usb_device_shutdown()”

3.1.8 _usb_device_init_endpoint()

Initializes an endpoint for the USB device controller.

Synopsis

```
uint_8 _usb_device_init_endpoint (
        _usb_device_handle handle,
        uint_8  endpoint_number,
        uint_16 max_packet_size,
        uint_8 direction,
        uint_8 endpoint_type,
        uint_8 flag)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number

max_packet_size[in] — Maximum packet size (in bytes) for the endpoint

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

USB_RECV

USB_SEND

endpoint_type [in] — Type of endpoint; one of:

USB_BULK_ENDPOINT

USB_CONTROL_ENDPOINT

USB_INTERRUPT_ENDPOINT

USB_ISOCHRONOUS_ENDPOINT

flag [in] — Zero size packet sent to terminate USB transfer

Description

This function initializes an endpoint by validating the parameters, initializing the software structures, and setting up the controller.

Return value

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

See also:

3.1.3 "_usb_device_deinit_endpoint()"

6.1.7 "USB_EP_STRUCT_PTR"

3.1.9 _usb_device_read_setup_data()

Reads the setup data for the endpoint.

Synopsis

```
void _usb_device_read_setup_data(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uchar_ptr 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.

Return Value

None

See Also:

3.1.6 "_usb_device_init()"

3.1.8 "_usb_device_init_endpoint()"

3.1.10 "_usb_device_recv_data()"

3.1.10 _usb_device_recv_data()

Receives data from the endpoint.

Synopsis

```
uint_8 _usb_device_recv_data(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uchar_ptr buffer_ptr,
        uint_32 size)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number for the transaction

buffer_ptr [out] — Pointer to the application buffer where the driver layer copies the data

size [in] — Number of bytes to receive

Description

The function calls this device layer API function to receive the data from the endpoint specified by *endpoint_number*. The function enqueues the receive request by passing data size as parameter along with the buffer pointer. If the data size to receive is smaller than the data available, the device layer then copies the requested data. Otherwise, the complete data when received must be sent using an event from the interrupt. 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 data is received.

Return value

- **USB_OK** (success)
- **USBERR_RX_FAILED** (failure: data reception from the endpoint failed)

See Also:

3.1.9 "_usb_device_read_setup_data()"

3.1.11 "_usb_device_register_service()"

6.1.8 "USB_PACKET_SIZE"

3.1.17 "_usb_device_unregister_service()"

3.1.11 _usb_device_register_service()

Registers the service for the type of event or endpoint.

Synopsis

```
uint_8 _usb_device_register_service(
        uint_8 controller_ID,
        uint_8 type,
        USB_SERVICE_CALLBACK service)
```

Parameters

controller_ID [in] — USB device controller ID

type [in] — Identifies the event on the bus or the endpoint number; can take the following values:

USB_SERVICE_BUS_RESET

USB_SERVICE_EP<n> — where *n* is the endpoint number the device supports

USB_SERVICE_ERROR

USB_SERVICE_RESUME

USB_SERVICE_SLEEP

USB_SERVICE_SOF

USB_SERVICE_SPEED_DETECTION

USB_SERVICE_STALL

USB_SERVICE_SUSPEND

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

Description

The function registers the callback service for an event or an endpoint if it has not been registered. When an event occurs at the bus or data is sent or received at an endpoint, the device layer calls the corresponding service callback function registered for this event.

Return Value

- **USB_OK** (success)
- **USBERR_ALLOC_SERVICE** — Service has already been registered

See Also:

6.1.10 "USB_SERVICE_CALLBACK()"

3.1.17 "_usb_device_unregister_service()"

3.1.12 _usb_device_send_data()

Sends data on the endpoint.

Synopsis

```
uint_8 _usb_device_send_data(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uchar_ptr buffer_ptr,
        uint_32 size)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — USB 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 this device layer API function to send the data on the endpoint specified by *endpoint_number*. The function enqueues the send request by passing data size as 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.

Return Value

- **USB_OK** (success)
- **USBERR_TX_FAILED** (failure: data transfer from the endpoint failed)

See Also:

3.1.10 "_usb_device_recv_data()"

3.1.11 "_usb_device_register_service()"

3.1.17 "_usb_device_unregister_service()"

6.1.8 "USB_PACKET_SIZE"

3.1.13 _usb_device_set_address()

Sets the address of the USB Device.

Synopsis

```
void _usb_device_set_address(
        _usb_device_handle handle,
        uint_8 address)
```

Parameters

handle [in] — USB Device handle

address [in] — Address of the USB device

Description

The function calls this device layer API function to initialize the device address and 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

None

3.1.14 _usb_device_set_status()

Sets the internal USB device state.

Synopsis

```
uint_8 _usb_device_set_status(
        _usb_device_handle handle,
        uint_8 component,
        uint_16 setting)
```

Parameters

handle [in] — USB Device handle

component [in] — Components defined for *USB Chapter 9* Get_Status/Set_Status calls, the possible values are:

USB_STATUS_ADDRESS

USB_STATUS_CURRENT_CONFIG

USB_STATUS_DEVICE

USB_STATUS_DEVICE_STATE

USB_STATUS_INTERFACE

USB_STATUS_SOF_COUNT

USB_STATUS_TEST_MODE

USB_STATUS_ENDPOINT — The LSB nibble carries the endpoint number

status [in] — Current status of the component

Description

The function calls this device layer API function to set the status of the defined component used for *USB Chapter 9* Get_Status or Set_Status USB framework calls. This call is usually made when the Set_Status USB framework call is received from the USB host.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: status of the defined component is not set)

See Also:

3.1.4 "_usb_device_get_status()"

3.1.15 _usb_device_shutdown()

Shuts down the USB device controller.

Synopsis

```
void _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.

The function does the following:

1. Terminates all transactions
2. Unregisters all the services
3. Disconnects the device from the USB bus

Return Value

None

See Also:

3.1.6 "_usb_device_init()"

3.1.16 _usb_device_stall_endpoint()

Stalls the endpoint in the specified direction.

Synopsis

```
void _usb_device_stall_endpoint(
_usb_device_handle handle,
uint_8 endpoint_number,
uint_8 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

Description

The function calls this device layer API function when:

- The application or class does not support a particular control function.
- The endpoint is not functioning properly.

Return Value

None

See Also:

3.1.18 "_usb_device_unstall_endpoint()"

3.1.17 _usb_device_unregister_service()

Unregisters the service for the type of event or endpoint.

Synopsis

```
uint_8 _usb_device_unregister_service(
        _usb_device_handle handle,
        uint_8 event_endpoint)
```

Parameters

handle [in] — USB Device handle

event_endpoint [in] — Identifies the event on the bus or the endpoint number; can take the following values:

USB_SERVICE_BUS_RESET

USB_SERVICE_EP<n> — where *n* is the endpoint number the device supports

USB_SERVICE_ERROR

USB_SERVICE_RESUME

USB_SERVICE_SLEEP

USB_SERVICE_SOF

USB_SERVICE_SPEED_DETECTION

USB_SERVICE_STALL

USB_SERVICE_SUSPEND

Description

The function unregisters 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.

Return Value

- **USB_OK** (success)
- **USBERR_UNKNOWN_ERROR** — Service has not been registered

See Also:

3.1.11 "_usb_device_register_service()"

3.1.18 _usb_device_unstall_endpoint()

Unstalls the endpoint in the specified direction.

Synopsis

```
void _usb_device_unstall_endpoint(
        _usb_device_handle handle,
        uint_8 endpoint_number,
        uint_8 direction)
```

Parameter

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number to unstall

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

USB_RECV

USB_SEND

Description

The function calls this device layer API function to unstall the endpoint after it has been previously stalled.

Return Value

None

See Also:

3.1.16 "_usb_device_stall_endpoint()"

Chapter 4
USB Device Class API

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

4.1 CDC Class API function listings

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

4.1.1 USB_Class_CDC_Init()

Initialize the CDC class.

Synopsis

```
uint_8 USB_Class_CDC_Init(
        uint_8 controller_ID,
        USB_CLASS_CALLBACK cdc_class_callback,
        USB_REQ_FUNC vendor_req_callback,
        USB_CLASS_CALLBACK pstn_callback)
```

Parameters

controller_ID [in] — USB device controller ID

cdc_class_callback [in] — Callback for generic events like TRANSPORT CONNECTED, DATA RECEIVED, DATA SENT, and so on

vendor_req_callback [in] — Callback function for vendor specific requests

pstn_callback [in] — Function specific callback function

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registerization failed)

See Also:

6.1.2 "USB_CLASS_CALLBACK()"

6.1.9 "USB_REQ_FUNC()"

4.1.2 USB_Class_CDC_DeInit()

De-initialize CDC class.

Synopsis

```
uint_8 USB_Class_CDC_DeInit (uint_8 controller_ID )
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the CDC class and controller hardware.

Return value

USB_OK(success)

4.1.3 USB_Class_CDC_Interface_CIC_Send_Data()

Send Communication Interface Class (CIC) data.

Synopsis

```
uint_8 USB_Class_CDC_Interface_CIC_Send_Data(
        uint_8 controller_ID,
        uint_8_ptr buff_ptr,
        USB_PACKET_SIZE size)
```

Parameters

controller_ID [in] — USB device controller ID

buff_ptr [in] —Pointer to the buffer containing data

size [in] — Size of data in the buffer

Description

The application calls this API function to send CIC data specified by *buff_ptr* and *size*. Data is sent over CIC_SEND_ENDPOINT 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. This function is used by sub-class implementation (for example, PSTN sub-class) to send notification data on INTERRUPT ENDPOINT (when CIC_NOTIF_ELEM_SUPPORT macro is enabled).

Return Value

- **USB_OK** (success)
- **USBERR_TX_FAILED** (failure: CIC_SEND_ENDPOINT is not defined)
- **USBERR_DEVICE_BUSY** (failure: Send Queue is full)

See Also:

6.1.8 "USB_PACKET_SIZE"

4.1.4 USB_Class_CDC_Interface_DIC_Recv_Data()

Receive Data Interface Class (DIC) data.

Synopsis

```
uint_8 USB_Class_CDC_Interface_DIC_Recv_Data (
        uint_8 controller_ID,
        uint_8_ptr buff_ptr,
        USB_PACKET_SIZE size)
```

Parameters

controller_ID [in] — USB device controller ID

buff_ptr [in] —Pointer to the buffer to receive data

size [in] — Size of data to be received

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 over DIC_RECV_ENDPOINT 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)
- **USBERR_RX_FAILED** (failure: callback event not received)

See Also:

6.1.8 "USB_PACKET_SIZE"

4.1.5 USB_Class_CDC_Interface_DIC_Send_Data()

Send Data Interface Class (DIC) data.

Synopsis

```
uint_8 USB_Class_CDC_Interface_DIC_Send_Data(
        uint_8 controller_ID,
        uint_8_ptr buff_ptr,
        USB_PACKET_SIZE size)
```

Parameters

controller_ID [in] — USB device controller ID

buff_ptr [in] —Pointer to the buffer containing data

size [in] — Size of data in the buffer

Description

The application calls this API function to send DIC data specified by *buff_ptr* and *size*. Data is sent over DIC_SEND_ENDPOINT 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)
- **USBERR_TX_FAILED** (failure: DIC_SEND_ENDPOINT is not defined)
- **USBERR_DEVICE_BUSY** (failure: Send Queue is full)

See Also:

6.1.8 "USB_PACKET_SIZE"

4.1.6 USB_Class_CDC_Periodic_Task()

Complete any left over activity on a specified time period.

Synopsis

```
void USB_Class_CDC_Periodic_Task(void)
```

Parameters

None

Description

The application calls this API function so the class driver can complete any left over activity. It is recommended to make this call on a timer context.

Return Value

None

4.2 HID Class API function listings

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

4.2.1 USB_Class_HID_Init()

Initialize the HID class.

Synopsis

```
uint_8 USB_Class_HID_Init(
        uint_8 controller_ID,
        USB_CLASS_CALLBACK hid_class_callback,
        USB_REQ_FUNC vendor_req_callback
        USB_CLASS_SPECIFIC_HANDLER_FUNC param_callback)
```

Parameters

controller_ID [in] — USB device controller to initialize

hid_class_callback [in] — Callback for generic events like ENUM COMPLETE, DATA RECEIVED, DATA SENT, and so on

vendor_req_callback [in] — Callback function for vendor specific requests

param_callback [in] — Callback for HID specific control requests

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registerization failed)

See Also:

6.1.2 "USB_CLASS_CALLBACK()"

6.1.5 "USB_CLASS_SPECIFIC_HANDLER_FUNC()"

6.1.9 "USB_REQ_FUNC()"

4.2.2 USB_Class_HID_DeInit()

De-initialize HID class.

Synopsis

```
uint_8 USB_Class_HID_DeInit (uint_8 controller_ID)
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the HID class and controller hardware.

Return value

USB_OK (success)

4.2.3 USB_Class_HID_Periodic_Task()

Complete any left over activity on a specified time period.

Synopsis

```
void USB_Class_HID_Periodic_Task(void)
```

Parameters

None

Description

The application calls this API function so the class driver can complete any left over activity. It is recommended to make this call on a timer context.

Return Value

None

4.2.4 USB_Class_HID_Send_Data()

Send HID data.

Synopsis

```
uint_8 USB_Class_HID_Send_Data(
        uint_8 controller_ID,
        unit 8 ep_num,
        uint_8_ptr buff_ptr,
        USB_PACKET_SIZE size)
```

Parameters

controller_ID [in] — USB device controller ID

ep_num [in] — USB endpoint number

buff_ptr[in] — Pointer to the buffer containing data

size [in] — Size of data in the buffer

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_TX_FAILED** (failure: HID report data has not been sent)

See Also:

6.1.8 "USB_PACKET_SIZE"

4.3 MSC Class API function listings

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

4.3.1 USB_Class_MSC_Init()

Initialize the MSC class.

Synopsis

```
uint_8 USB_Class_MSC_Init (
        uint_8 controller_ID,
        USB_CLASS_CALLBACK msc_class_callback,
        USB_REQ_FUNC vendor_req_callback,
        USB_CLASS_CALLBACK param_callback);
```

Parameters

controller_ID [in] — USB device controller ID

msc_class_callback [in] — Callback for generic events like USB_APP_ENUM_COMPLETE and so on

vendor_req_callback [in] — Callback function for vendor specific requests

param_callback [in] — Function specific callback function

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registerization failed)

See Also:

6.1.2 "USB_CLASS_CALLBACK()"

6.1.9 "USB_REQ_FUNC()"

4.3.2 USB_Class_MSC_DeInit()

De-initialize MSC class.

Synopsis

```
uint_8 USB_Class_MSC_DeInit (uint_8 controller_ID)
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the MSC class and controller hardware.

Return value

USB_OK (success)

4.3.3 USB_Class_MSC_Periodic_Task()

Complete any left over activity on a specified time period.

Synopsis

```
void USB_MSC_Periodic_Task (void)
```

Parameters

None

Description

The application calls this API function so the class driver can complete any left over activity. It is recommended to make this call on a timer context.

Return Value

None

4.4 PHDC Class API function listings

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

4.4.1 USB_Class_PHDC_Init()

Initialize the PHDC class.

Synopsis

```
uint_8 USB_Class_PHDC_Init(
        uint_8 controller_ID,
        USB_CLASS_CALLBACK phdc_class_callback,
        USB_REQ_FUNC vendor_req_callback)
```

Parameters

controller_ID [in] — USB device controller ID

phdc_class_callback [in] — Callback for generic events like TRANSPORT CONNECTED, DATA RECEIVED, DATA SENT, and so on

vendor_req_callback [in] — Callback function for vendor specific requests

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registerization failed)

See Also:

6.1.2 "USB_CLASS_CALLBACK()"

6.1.9 "USB_REQ_FUNC()"

4.4.2 USB_Class_PHDC_DeInit()

De-initialize PHDC class.

Synopsis

```
uint_8 USB_Class_PHDC_DeInit (uint_8 controller_ID)
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the PHDC class and controller hardware.

Return value

USB_OK (success)

4.4.3 USB_Class_PHDC_Periodic_Task()

Complete any left over activity on a specified time period.

Synopsis

```
void USB_Class_PHDC_Periodic_Task(void)
```

Parameters

None

Description

The application calls this API function so the class driver can complete any left over activity. It is recommended to make this call on a timer context.

Return Value

None

4.4.4 USB_Class_PHDC_Send_Data()

Send PHDC data.

Synopsis

```
uint_8 USB_Class_PHDC_Send_Data(
        uint_8 controller_ID,
        boolean meta_data,
        unit 8 num_tfr,
        unit 8 current_qos,
        uint_8_ptr app_buff,
        USB_PACKET_SIZE size);
```

Parameters

controller_ID [in] — USB device controller ID

meta_data [in] — Opaque metadata in application buffer or not

num_tfr [in] — Number of transfers to follow with given channel, only valid if metadata is true

current_qos [in] —Quality of Service (QOS) of the transfers to follow, only valid if metadata is true

app_buff [in] —Pointer to the buffer containing data

size [in] — Size of data in the buffer

Description

The function calls this API function to send PHDC report data specified by *meta_data*, *num_tfr*, *current_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)
- **USBERR_TX_FAILED** (failure: PHDC report data has not been sent)

See Also:

6.1.8 "USB_PACKET_SIZE"

4.4.5 USB_Class_PHDC_Recv_Data()

Receives data from the PHDC Receive Endpoint of desired QOS.

Synopsis

```
uint_8 USB_Class_PHDC_Recv_Data(
        uint_8 controller_ID,
        unit 8 current_qos,
        uint_8_ptr buff_ptr,
        USB_PACKET_SIZE size);
```

Parameters

controller_ID [in] — USB device controller ID

current_qos [in] — QOS

buff_ptr [out] —Pointer to the application buffer where the driver layer copies the data

size [in] — Size of data in the buffer

Description

The function is used to receive PHDC data from the endpoint specified by *current_qos*. This function uses 3.1.10 "_usb_device_recv_data()" function to perform the required functionality.

Return Value

- **USB_OK** (success)
- **USBERR_RX_FAILED** (failure: data reception from the endpoint failed)

See Also:

3.1.10 "_usb_device_recv_data()"

6.1.8 "USB_PACKET_SIZE"

4.5 USB Audio Device Class API function listings

4.5.1 USB_Class_Audio_Init()

Initializes the audio class and the controller hardware.

Synopsis

```
uint_8 USB_Class_Audio_Init(
        uint_8 ControllerID,USB_CLASS_CALLBACK audio_class_callback,
        USB_REQ_FUNC vendor_req_callback,
        USB_CLASS_CALLBACK param_callback)
```

Parameters

ControllerID [in]—USB device controller ID

audio_class_callback [in]—Callback for generic events like TRANSPORT CONNECTED, DATA RECEIVED, DATA SENT, and so on

vendor_req_callback [in]—Callback function for vendor specific requests

param_callback [in]—Callback for Audio specific control requests

Description

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

Return Value

- **USB_OK** (success)
- **USBERR__INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registration failed)

See Also

6.1.2 “USB_CLASS_CALLBACK()”

6.1.9 “USB_REQ_FUNC()”

6.1.5 “USB_CLASS_SPECIFIC_HANDLER_FUNC()”

4.5.2 USB_Class_Audio_DeInit()

De-initialize Audio class.

Synopsis

```
uint_8 USB_Class_Audio_DeInit (uint_8 controller_ID)
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the Audio class and controller hardware.

Return value

USB_OK (success)

4.5.3 USB_Class_Audio_Send_Data()

Sends audio data to the application layer as specified in the app_buff.

Synopsis

```
uint_8 USB_Class_Audio_Send_Data(uint_8 ControllerID, uint_8 ep_num,
                                 uint_8_ptr app_buff,USB_PACKET_SIZE size)
```

Parameters

controllerID [in]—USB device controller ID

ep_num [in]—USB endpoint number

app_buff [in]—Pointer to the buffer containing data

size [in]—Size of data in the buffer

Description

This API function is called to send Audio data specified by app_buff, size. Once the data has been sent, the application layer receives a callback event. The application is to maintain the transmit buffer reserved until it receives a callback event stating that data has been sent. The Audio class is not buffering the transmit data internally.

Return Value

- **USB_OK** (success)
- **USBERR_TX_FAILED** (failure: the data has not been sent)
- **USBERR_DEVICE_BUSY** (failure: send queue is full)

4.5.4 USB_Audio_Class_Recv_Data()

Receives data as specified by the pointer to the application buffer.

Synopsis

```
uint_8 USB_Class_Audio_Recv_Data(uint_8 controllerID, uint_8 ep_num,
                                 uint_8_ptr app_buff, USB_PACKET_SIZE size)
```

Parameters

controllerID [in]—USB device controller ID

ep_num [in]—Endpoint number

app_buff [in]—Pointer to the application buffer where the driver layer copies the data

size [in]—Size of data in the buffer

Description

This API function is called to receive data specified in buff_ptr and length given by size. 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)
- **USBERR_TX_FAILED** (failure: The data has not been received)

See Also

6.1.8 “USB_PACKET_SIZE"

4.6 USB DFU Device Class API function listings

The DFU class provides a DFU control interface to controls the functional behavior of particular DFU function.

4.6.1 USB_Class_DFU_Init()

Initialize the DFU class.

Synopsis

```
uint_8 USB_Class_DFU_Init
(
      uint_8 ControllerID,
      USB_CLASS_CALLBACK dfu_class_callback,
      USB_REQ_FUNC vendor_req_callback,
      USB_CLASS_SPECIFIC_HANDLER_FUNC param_callback
);
```

Parameters

ControllerID [IN]–USB device controller ID.

dfu_class_callback [IN]–Callback for generic events like USB_APP_ENUM_COMPLETE and so on.

vendor_req_callback [IN]–Callback function for vendor specific requests.

param_callback [IN]–Callback for DFU specific callback function

Description

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

Return Value

- **USB_OK** (success)
- **USBERR__INVALID_NUM_OF_ENDPOINTS** (failure: endpoints defined are greater than the supported number for the device)
- **USBERR_EP_INIT_FAILED** (failure: device initialization failed)
- **USBERR_ALLOC_SERVICE** (failure: callback registration failed)

See Also:

6.1.2, “USB_CLASS_CALLBACK()”

6.1.9, “USB_REQ_FUNC()”

6.1.5, “USB_CLASS_SPECIFIC_HANDLER_FUNC()”

4.6.2 USB_Class_DFU_DeInit()

De-initialize DFU class.

Synopsis

```
uint_8 USB_Class_DFU_DeInit (uint_8 controller_ID)
```

Parameter

controller_ID [in] — USB device controller ID

Description

The application calls this API function to de-initialize the DFU class and controller hardware.

Return value

USB_OK (success)

4.6.3 USB_Class_DFU_Periodic_Task()

Complete any left over activity on a specified time period.

Synopsis

```
void USB_Class_DFU_Periodic_Task(void)
```

Parameters

None

Description

The application calls this API function to implement flashing and manifesting processes.

Return Value

None

4.7 USB Battery Charging API

This section defines the API functions used for the Battery charging. The application can use these API functions to make Battery Charging applications.

4.7.1 _usb_batt_chg_init()

The USB Battery Charging initialization function is used to initialize the USBDCD module according to the initialization parameters passed in.

Synopsis

```
uint_32 _usb_batt_chg_init( BATT_CHG_INIT_STRUCT* init_struct);
```

Parameters

init_struct [in]—Pointer to the provided application initialization parameters

Description

This function will allocate the memory necessary for the battery charging driver operation, initialize the sequence state machine and configure the USBDCD registers according to the input arguments.

If his operation is successful, this function will return USB_OK otherwise different error status listed bellow, will be returned.

Return Value

- **USB_OK** (Driver initialization was successfully performed)
- **USB_INVALID_PARAMETER** (Wrong parameters passed in)
- **USBERR_ALLOC** (Driver memory allocation failed)

4.7.2 _usb_batt_chg_uninit()

Uninitialize the Battery Charging driver.

Synopsis

```
uint_32 _usb_batt_chg_uninit(void);
```

Parameters

None

Description

This function will de-allocate the memory necessary for the battery charging driver operation and will set the pointer to that memory area, to NULL.

Return Value

- **USB_OK** (Driver un-initialization was successfully performed.)
- **USB_INVALID_PARAMETER** (Pointer to the memory structure is already NULL)

4.7.3 _usb_batt_chg_register_callback()

The USB Battery Charging register callback function is used to initialize the application defined callback function that the driver will call at the completion of the sequence phases.

Synopsis

```
uint_32 _usb_batt_chg_register_callback(usb_batt_chg_callback callback);
```

Parameters

callback [in]—The pointer to the callback function to be registered.

Description

This function will save the user provided callback function pointer into the internal Battery charging structure. This function will be used to signal to the application, the corresponding event at the completion of the sequence phases.

Return Value

- **USB_OK** (Callback registration successfully performed)
- **USB_INVALID_PARAMETER** (NULL pointer provided for the handle.)

4.7.4 _usb_batt_chg_task()

Battery Charging driver task.

Synopsis

```
void _usb_batt_chg_task(void);
```

Parameters

None

Description

The USB Battery charging task function is used to service the registered callback function as long as any battery charging event is pending. After the callback function completion, the event is cleared.

Notice that the task shall be called by the application together with the rest of the tasks, in the main sequence loop.

Return Value

None

4.7.5 _usb_batt_chg_ext_isr()

Enables battery charging VBUS state change external interrupt function.

Synopsis

```
void _usb_batt_chg_ext_isr(void);
```

Parameters

None

Description

This function is used by the application layer to inform the battery charging driver that the change on the voltage VBUS line has occurred. This detection is done outside of the battery charging context and is realized by an external IC that can be the external OTG status monitor IC.

Return Value

None

4.8 USB Video Device Class API function listing

4.8.1 USB_Class_Video_Init()

Initializes the Video class.

Synopsis

```
uint_8 USB_Class_Video_Init
(
      uint_8 ControllerID,
      USB_CLASS_CALLBACK Video_class_callback,
      USB_REQ_FUNC vendor_req_callback,
      USB_CLASS_SPECIFIC_HANDLER_FUNC param_callback
)
```

Parameters

ControllerID [IN]—USB device controller ID.

Video_class_callback [IN]—Callback for generic events like TRANSPORT CONNECTED, DATA RECEIVED, DATA SENT, and so on.

vendor_req_callback [IN]—Callback function for vendor specific requests.

param_callback [IN]—Callback for Video specific control requests.

Description

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

Return Value

USB_OK (success)

USBERR_INVALID_NUM_OF_ENDPOINTS (failure: endpoints defined are greater than the supported number for the device)

USBERR_EP_INIT_FAILED (failure: device initialization failed)

USBERR_ALLOC_SERVICE (failure: callback registration failed)

See Also:

6.1.2, “USB_CLASS_CALLBACK()”

6.1.9, “USB_REQ_FUNC()”

6.1.5, “USB_CLASS_SPECIFIC_HANDLER_FUNC()”

4.8.2 USB_Class_Video_Send_Data()

Synopsis

```
uint_8 USB_Class_Video_Send_Data
   (
      uint_8 ControllerID,
      uint_8 ep_num,
      uint_8_ptr app_buff,
      uint_32 size
   )
```

Parameters

controllerID [IN]—USB device controller ID.

ep_num [IN]—USB endpoint number.

app_buff [IN]—Pointer to the buffer containing data.

size [IN]—Size of data in the buffer.

Description

The function calls this API function to send Video data specified by app_buff, size. Once the data has been sent, the application layer receives a callback event. The application is to maintain the transmit buffer reserved until it receives a callback event stating that data has been sent. The Video class is not buffering the transmit data internally.

Return Value

USB_OK (success)

USBERR_TX_FAILED (failure: The data has not been sent)

USBERR_DEVICE_BUSY (failure: Send queue is full)

Chapter 5
USB Descriptor API

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

5.1 USB Descriptor API function listings

5.1.1 USB_Desc_Get_Descriptor()

Gets various descriptors from the application.

Synopsis

```
uint_8 USB_Desc_Get_Descriptor(
        uint_8 controller_ID,
        uint_8 type,
        uint_8 str_num,
        uint_16 index,
        uint_8_ptr *descriptor,
        USB_PACKET_SIZE *size)
```

Parameters

controller_ID [in] — USB device controller ID

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

index [out] — Size of descriptor returned

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_REQ_TYPE** — Invalid request

Sample Implementation

```
uint_8 USB_Desc_Get_Descriptor(
     uint_8 controller_ID,   /* [IN] controller ID */
     uint_8 type,            /* [IN] type of descriptor requested */
     uint_8 str_num,         /* [IN] string index for string descriptor */
     uint_16 index,          /* [IN] string descriptor language Id */
```

```
        uint_8_ptr *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 = (uint_8_ptr)g_std_descriptors [type];
              *size = g_std_desc_size[type];
            }
            break;
          case USB_HID_DESCRIPTOR:
            {
              type = USB_CONFIG_DESCRIPTOR ;
              *descriptor = (uint_8_ptr)(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 = (uint_8_ptr)g_languages.languages_supported_string;
                    *size = g_languages.languages_supported_size;
                } else
                {
                    uint_8 lang_id=0;
                    uint_8 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 =
              (uint_8_ptr)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 = (uint_8_ptr)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 used and their properties.

Synopsis

```
void *USB_Desc_Get_Endpoints(
        uint_8 controller_ID)
```

Parameters

controller_ID [in] — USB device controller ID

Description

The class driver calls this function to the application to get information on all 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(
      uint_8 controller_ID      /* [IN] Controller ID */
)
{
    #pragma unused (controller_ID)
    return (void*)&usb_desc_ep;
}
```

See Also:

6.1.4 "USB_CLASS_PHDC_CHANNEL_INFO"

6.1.6 "USB_ENDPOINTS"

5.1.3 USB_Desc_Get_Interface()

Gets the currently configured interface.

Synopsis

```
uint_8 USB_Desc_Get_Interface(
        uint_8 controller_ID,
        unit 8 interface,
        uint_8_ptr alt_interface)
```

Parameters

controller_ID [in] — USB device controller ID

interface [in] — Interface number

alt_interface [out] — Output alternate interface

Description

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

Return Value

- **USB_OK** (success)
- **USBERR_INVALID_REQ_TYPE** — Invalid request

Sample Implementation

```
uint_8 USB_Desc_Get_Interface(
      uint_8 controller_ID, /* [IN] controller Id */
      uint_8 interface,     /* [IN] interface number */
      uint_8_ptr 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 wakeup or not.

Synopsis

```
boolean USB_Desc_Remote_Wakeup(uint_8 controller_ID)
```

Parameters

controller_ID [in] — USB device controller ID

Description

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

Return Value

- **TRUE** (Remote wakeup supported)
- **FALSE** (Remote wakeup not supported)

Sample Implementation

```
boolean USB_Desc_Remote_Wakeup(
      uint_8 controller_ID  /* [IN] controller Id */
)
{
    return REMOTE_WAKEUP_SUPPORT;
}
```

5.1.5 USB_Desc_Set_Interface()

Sets new interface.

Synopsis

```
uint_8 USB_Desc_Set_Interface(
        uint_8 controller_ID,
        unit 8 interface,
        uint_8 alt_interface)
```

Parameters

controller_ID [in] — USB device controller ID

interface [in] — Interface number

alt_interface [in] — Input alternate interface

Description

The framework module calls this function to the application 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** — Invalid request

Sample Implementation

```
uint_8 USB_Desc_Set_Interface(
      uint_8 controller_ID, /* [IN] controller Id */
      uint_8 interface,     /* [IN] interface number */
      uint_8 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

```
boolean USB_Desc_Valid_Configation(
        uint_8 controller_ID,
        unit_16 config_val)
```

Parameters

controller_ID [in] — USB device controller ID

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

```
boolean USB_Desc_Valid_Configation(
      uint_8 controller_ID,/*[IN] controller Id */
      uint_16 config_val   /*[IN] configuration value */
)
{
    uint_8 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

```
boolean USB_Desc_Valid_Interface(
        uint_8 controller_ID,
        unit_8 interface)
```

Parameters

controller_ID [in] — USB device controller ID

interface [in] — USB descriptor target interface

Description

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

Return Value

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

Sample Implementation

```
boolean USB_Desc_Valid_Interface(
      uint_8 controller_ID,/*[IN] controller Id */
      uint_8 interface   /*[IN] target interface */
)
{
    uint_8 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 the various API functions.

6.1 Data structure listings

6.1.1 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
{
        uint_8 controller_ID,
        uint_8 ep_num;
        boolean setup;
        boolean direction;
        uint_8_ptr buffer_ptr;
        USB_PACKET_SIZE len;
        uint_8 errors;
}USB_EVENT_STRUCT, *PTR_USB_EVENT_STRUCT;
```

Fields

controller_ID — USB controller ID

ep_num — USB endpoint number

setup — *buffer_ptr* contains setup packet or not

direction — Direction of endpoint, one of:

USB_RECV

USB_SEND

buffer_ptr — Buffer containing data

size — Size of data buffer

errors — USB error code

6.1.2 USB_CLASS_CALLBACK()

This callback function is called for generic events and is passed as an input parameter to 4.2.1 “USB_Class_HID_Init()”, 4.4.1 “USB_Class_PHDC_Init()”, 4.5.1 “USB_Class_Audio_Init()”, 4.6.1, “USB_Class_DFU_Init()”, and Section 4.8.1, “USB_Class_Video_Init()” from the application to the class driver. The “type” input parameter states the type of event. 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 implementing 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)(
        uint_8 controller_ID,
        uint_8 type,
        void* data);
```

Callback Parameters

controller_ID — USB controller ID

type — Type of event

data — Event data based on the type value

6.1.3 USB_CLASS_PHDC_CHANNEL

This structure defines information about the non-control endpoints used by the PHDC application using *channel_num* as parameter.

Synopsis

```
typedef struct _usb_class_phdc_channel
{
    uint_8 channel_num;
    uint_8 type;
    uint_8 direction;
    USB_PACKET_SIZE size;
    uint_8 qos;
}USB_CLASS_PHDC_CHANNEL;
```

Fields

channel_num — PHDC channel number

type — Type of the endpoint; one of :

USB_BULK_PIPE

USB_CONTROL_PIPE

USB_INTERRUPT_PIPE

direction — Direction of the endpoint; one of :

USB_RECV

USB_SEND

size — Size of buffer to be used at the device layer

qos — Quality of service supported

6.1.4 USB_CLASS_PHDC_CHANNEL_INFO

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

Synopsis

```
typedef const struct _usb_class_phdc_channel_info
{
    uint_8 count;
    USB_CLASS_PHDC_CHANNEL channel[PHDC_DESC_ENDPOINT_COUNT];
}USB_CLASS_PHDC_CHANNEL_INFO, *PTR_USB_CLASS_PHDC_CHANNEL_INFO;
```

Fields

count — Count of non-control endpoints

channel — Properties of each PHDC channel

6.1.5 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 uint_8 (_CODE_PTR_ USB_CLASS_SPECIFIC_HANDLER_FUNC)(
        uint_8 request,
        uint_16 value,
        uint_8_ptr *buff,
        USB_PACKET_SIZE *size);
```

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

Callback Parameters

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

6.1.6 USB_ENDPOINTS

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

Synopsis

```
typedef const struct _USB_ENDPOINTS
{
    uint_8 count;
    USB_EP_STRUCT ep[HID_DESC_ENDPOINT_COUNT];
}USB_ENDPOINTS;
```

Fields

count — Count of non-control endpoints

ep — Properties of each endpoint

6.1.7 USB_EP_STRUCT_PTR

This structure defines parameters that are passed to 3.1.8 "_usb_device_init_endpoint()" API function to initialize a particular endpoint.

Synopsis

```
typedef struct _USB_EP_STRUCT
{
    uint_8 ep_num;
    uint_8 type;
    uint_8 direction;
    USB_PACKET_SIZE 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 in the device layer

6.1.8 USB_PACKET_SIZE

This macro defines what bit size must be used in size parameters throughout the stack. Stack requires more memory when higher width size data type is used and it also increases the binary size of the image built.

NOTE

For Audio and DFU class, a minimum of 16-bit data type should be used.

Synopsis

`#define USB_PACKET_SIZE <data type>` data type can be `uint_8`, `uint_16` or `uint_32`.

It is advised to use 8-bit data type for silicons with low memory and flash size.

6.1.9 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 uint_8 (_CODE_PTR_ USB_REQ_FUNC)(
        uint_8 controller_ID,
        USB_SETUP_STRUCT *setup_packet,
        uint_8_ptr *buff,
        USB_PACKET_SIZE *size);
```

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

Callback Parameters

controller_ID — USB controller ID

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

6.1.10 USB_SERVICE_CALLBACK()

This structure is used to represent the service callback functions registered to the device layer using the 3.1.11 "_usb_device_register_service()" call.

Synopsis

```
typedef void(_CODE_PTR_ USB_SERVICE_CALLBACK)(PTR_USB_EVENT_STRUCT);
```

Callback Parameters

PTR_USB_EVENT_STRUCT — This is a pointer to the structure containing information about the event occurred for which the corresponding service function is called.

6.1.11 USB_SETUP_STRUCT

This structure is passed as a parameter to the 6.1.9 "USB_REQ_FUNC() function and contains information about requests.

Synopsis

```
typedef struct _USB_SETUP_STRUCT
{
uint_8 request_type,
uint_8 request,
uint_16 value,
uint_16 index,
uint_16 length
}USB_SETUP_STRUCT;
```

Fields

request_type — Type of request

request — Request code

index — Value depends on which entity is addressed. The interface or endpoint is addressed in a low byte and the entity ID or zero in the high byte.

length — Length of the data

6.2 Battery charging data structure listings

6.2.1 USB_BATT_CHG_TIMING

This structure stores the battery charging timing list according to the Battery Charging specification.

Synopsis

```
typedef struct ubs_batt_chg_timings
{
  uint_16                   time_dcd_dbnc;
  uint_16                   time_vdpsrc_on;
  uint_16                   time_vdpsrc_con;
  uint_16                   time_seq_init;
  uint_8                    time_check_d_minus;
} USB_BATT_CHG_TIMINGS;
```

Fields

time_dcd_dbnc — Specifies the time period to debounce the D+ signal during the data pin contact phase (TDCD_DBNC)

time_vdpsrc_on — Indicates the time period for comparator enable , TVDPSRC_ON (D+ voltage source on)

time_vdpsrc_con — Time period before enabling the D+ pullup resistor, TVDPSRC_CON

time_seq_init — Sequence initiation time variable

time_check_d_minus — Time before check of the D– line

6.2.2 USB_BATT_CHG_INIT_STRUCT

This structure contains the Battery Charging initialization elements passed by the application at the initialization step.

Synopsis

```
typedef struct usb_batt_chg_init_struct
{
  boolean  ext_vbus_detect_circuit_use;
 ext_enable_disable_func   ext_vbus_detect_enable_disable_func;
  ext_vbus_det_get_status ext_vbus_detect_update_vbus_status_func;
  boolean  ext_batt_chg_circuit_use;
  ext_enable_disable_func ext_batt_chg_circuit_enable_disable_func;
  USB_BATT_CHG_TIMINGS usb_batt_chg_timmings_config;
} USB_BATT_CHG_INIT_STRUCT;
```

Fields

ext_vbus_detect_circuit_use—Boolean indicating the presence of the VBUS detect external IC

ext_vbus_detect_enable_disable_func—Pointer to the function responsible for enabling/disabling the VBUS detect external IC

ext_vbus_detect_update_vbus_status_func—Function used to record any change updates with the VBUS voltage status

ext_batt_chg_circuit_use—Boolean indicating the presence of the battery management external IC

ext_batt_chg_circuit_enable_disable_func—Pointer to the function responsible for enabling/disabling the battery management external IC

usb_batt_chg_timmings_config—Initial timing used to configure the battery charging driver driver

6.2.3 USB_BATT_CHG_STATUS

This structure stores the status elements which are passed by the driver to the registered application callback (device state, charger port type, error type a.s.o).

Synopsis

```
typedef struct usb_batt_chg_status
{
  uint_32         dev_state;
  boolean         vbus_valid;
  uint_32         charger_type;
  boolean         data_pin_det;
  error_type_t    error_type;

} USB_BATT_CHG_STATUS;
```

Fields

dev_state—Indicates the state of the sequencer

vbus_valid—Indicates if there is or not the voltage on the VBUS line

charger_type—Indicates the type of charging port according to the Battery Charging specification

data_pin_det—Indicates if the data pin detection has been done or not

error_type—If the port detection has failed, it specifies the type of the error (sequence period timeout, unknown port type)

6.2.4 USB_BAT_CHG_STRUCT

Battery charging internal structure used by the driver during operation and it contains the timing, status and event structures.

Synopsis

```
typedef struct usb_batt_chg_stuct
{
  USB_BATT_CHG_INIT       usb_batt_chg_init;
  USB_EVENT_STRUCT    usb_batt_chg_event;
  USB_BATT_CHG_STATUS      usb_batt_chg_status;
  usb_batt_chg_callback    app_callback;
} USB_BATT_CHG_STRUCT;
```

Fields

usb_batt_chg_init—Specifies the initialization structure member

usb_batt_chg_event—Specifies the battery charging event type (sequence phase complete, eror)

usb_batt_chg_status—Specifies the status member of the structure

app_callback—Pointer to the registered application callback function

Chapter 7
Reference Data Types

7.1 Data Types for Compiler Portability

Table 7-1. S08 Compiler Portability Data Types

Name	Bytes	Range		Description
		From	To	
boolean	1	0	NOT 0	0 = FALSE Non-zero = TRUE
uint_8	1	0	255	Unsigned character
uint_8_ptr	2	0	0xffff	Pointer to **uint_8**
uint_16	2	0	(2^16)–1	Unsigned 16-bit integer
uint_16_ptr	2	0	0xffff	Pointer to **uint_16**
uint_32	4	0	(2^32)–1	Unsigned 32-bit integer
uint_32_ptr	2	0	0xffff	Pointer to **uint_32**

Table 7-2. ColdFire V1 and V2 Compiler Portability Data Types

Name	Bytes	Range		Description
		From	To	
boolean	1	0	NOT 0	0 = FALSE Non-zero = TRUE
uint_8	1	0	255	Unsigned character
uint_8_ptr	4	0	0xffffffff	Pointer to **uint_8**
uint_16	2	0	(2^16)–1	Unsigned 16-bit integer
uint_16_ptr	4	0	0xffffffff	Pointer to **uint_16**

Table 7-2. ColdFire V1 and V2 Compiler Portability Data Types (continued)

Name	Bytes	Range		Description
		From	To	
uint_32	4	0	(2^32)–1	Unsigned 32-bit integer
uint_32_ptr	4	0	0xffffffff	Pointer to **uint_32**