USB Device Low Level Driver Interface Specification

Version 3.00

For use with USB Device Base System versions 3.14 and above

Date: 24-Oct-2014 18:24

Table of Contents

System Overview 4
Introduction 4
Packages and Documents 5
Packages 5
Documents 5
Driver Operation 6
Low Level Driver Components 6
Bus Event Monitor 6
EP0 Receive Handler 7
EP Receive/Transmit Handler 7
SOF Interrupt Handler (optional) 7
SOF Callback Table 7
Non-isochronous Endpoint Communication 8
Isochronous Endpoint Communication 8
Driver Configuration 9
Endpoint Information Management 10
API 11
Module Management 11
usbd_hw_init 11
usbd_hw_start 12
usbd_hw_stop 13
usbd_hw_delete 14
Driver Management 15
usbd_add_ep 15
usbd_remove_ep 16
usbd_send 17
usbd_receive 18
usbd_add_fifo 19
usbd_drop_fifo 20
usbd_at_high_speed 21
usbd_get_stall 22
usbd_set_stall 23
usbd_clr_stall 24
usbd_get_state 25
usbd_set_addr_post 26
usbd_set_addr_pre 27
usbd_set_cfg_post 28
usbd_set_cfg_pre 29
usbd_set_testmode 30
Callback Functions 31
usbd_pup_on_off 31
usbd_is_self_powered 32

usbd_conn_state 33
usbd_vbus 34
Error Codes 35
Types and Definitions 36
usbd_iso_buffer_t 36
USB Driver State Values 36
USB Transfer State Values 36
Test Modes 37
Events 37
Endpoint Attributes 37
Integration 38
OS Abstraction Layer (OAL) 38
PSP Porting 38

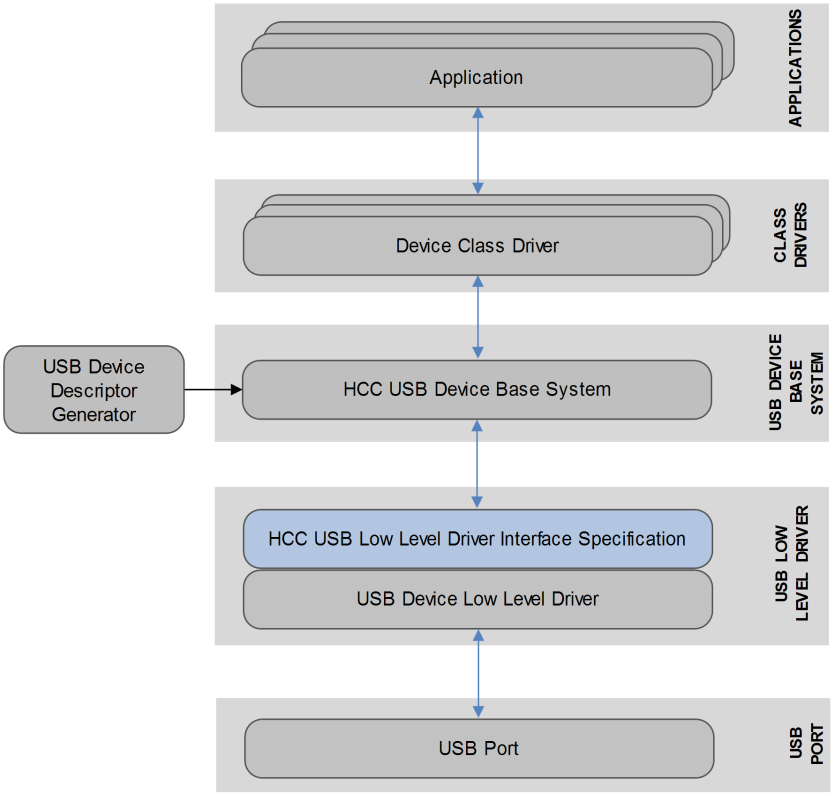
1 System Overview

1.1 Introduction

This guide is for those who want to implement a low level driver for HCC's Embedded USB Device (EUSBD) stack. There is no standard for implementing USB device interfaces and every USB device implementation is different. This document describes the low level driver interface that a driver for a USB device controller must provide to interoperate correctly with HCC's EUSBD stack.

Note: Users of the EUSBD system who are not implementing a low level driver should have no need to use or understand this document. If you do experience problems with the low level driver provided, please contact support@hcc-embedded.com.

This diagram shows the relationships between the main elements of the EUSBD stack:



A driver must be written specifically for a particular USB device controller. Each implementation is designed to work with the USB device core (the base system).

Note: Any engineer implementing a new low level driver should have a reasonable knowledge of the USB protocol, should understand the EUSBD system, and should be familiar with this document. In addition, a thorough understanding of the USB device controller is essential.

Note: We recommend contacting HCC before implementing a new driver. HCC can provide a tested driver for your target device, or at least one that is similar to it. HCC has developed low level drivers for many devices. It is unusual to find a USB device that HCC does not have experience with.

1.2 Packages and Documents

Packages

The following table lists the packages that you need in order to use this module.

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
usbd_base	The USB Device base package which includes the low level driver code. The USB Device Descriptor Generator is in the directory **hcc/util/configtool**.

Documents

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide
This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide
This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

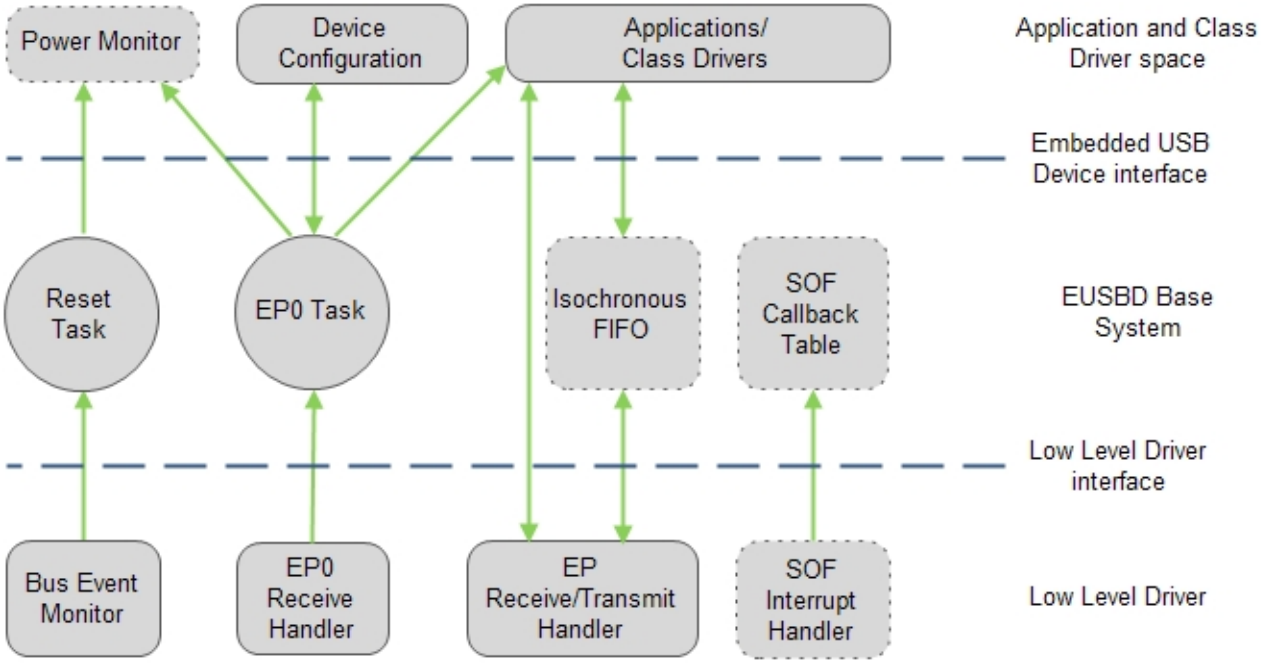
HCC Embedded USB Device Base System User's Guide

This document describes the Embedded USB Device base system.

HCC USB Device Low Level Driver Interface Specification
This is this document.

2 Driver Operation

The diagram below shows the whole EUSBD system. The components which are optional are shown with dotted borders. This document covers the low level driver and, in particular, the interface it provides to the rest of the EUSBD system.



2.1 Low Level Driver Components

Bus Event Monitor

The driver stack sets the **usbd_bus_event** if the status of the USB bus changes. The possible state changes are:

- USB suspended.
- USB wakeup.
- USB reset.

Note: In the suspended state the hardware may draw only limited current from the USB bus. Please refer to the Universal Serial Bus specification for full details.

EP0 Receive Handler

Endpoint 0 (referred to as "EP0") on a USB interface has special significance. The host sends packets known as "setup packets" (STPs) to this endpoint for configuration and control purposes. A setup packet requires special processing because there are timing constraints on the handling of certain types of STPs. For each setup packet received on EP0, an event is sent to the EP0 task (see Non-isochronous Endpoint Communication).

EP Receive/Transmit Handler

Data is transferred in blocks over bulk, interrupt, or control endpoints. Latency is not the most important factor when using this transfer type. Any delay during the transfer is handled using flow control on the USB bus.

All such USB transfers are described by a transfer descriptor *(usbd_transfer_t)*. Transfer descriptors are allocated by the class driver. After the transfer descriptor is filled, it is passed to **usbd_transfer()** if the transfer should not be blocked, or to **usbd_transfer_b()** if it should be blocked.

If the transfer is blocking, the call returns only when the transfer finishes successfully or returns an error.

If the transfer is non-blocking, the application must call **usbd_transfer_status()** periodically to allocate CPU time to transfer management, and to check the status of the transfer.

SOF Interrupt Handler (optional)

Implementation of this handler is optional. For most USB device controllers it is possible to get an interrupt every time a Start Of Frame (SOF) packet is received by the device. On a full speed bus the period is 1ms, on a high speed bus the period is 125µs.

Each time the SOF interrupt occurs, all the callback functions in the SOF callback table (see below) are executed. Note that these functions are called directly from the interrupt, so they are executed in the interrupt context.

There are three types of communication between the low level driver and the EUSBD stack, as follows:

- Non-isochronous endpoint communication.
- Isochronous endpoint communication.
- Bus event communication.

SOF Callback Table

The SOF callback table is a table of callback functions registered using functions defined in the *HCC Embedded USB Device Base System User's Guide*. Callback functions registered in this table define a period. If this period elapses, the callback becomes due. Due times are recalculated each time an SOF interrupt is generated. Note that, since these functions are called directly from the interrupt, they are executed in the interrupt context.

2.2 Non-isochronous Endpoint Communication

The base system, the USB driver common code, uses the transfer descriptor *usbd_transfer_t* to tell the low level driver the details of the transfer. This structure is passed to **usbd_send()** or **usbd_receive()**, which configures the endpoint according to the content of the structure.

Control endpoints need special handling. The stack supports only one control endpoint, EP0. EP0 must be created by the low level driver when **usbd_hw_start()** is called. EP0 must always be in a state in which it can receive the next setup packet. Thus, event generation for the reception of a setup packet must always be active. Also the USBDHW must always be able to receive an early handshake to abort an ongoing IN transfer.

The low level driver uses the *usbd_transfer_t::state* and *usbd_transfer_t::event* members to ask the base system to perform processing. When a transfer needs a status update (because an error occurred or the last chunk of data has been received or sent), it sets the state and, if the event field is not NULL, it calls **os_event_set_int()** on it.

The transfer state is set as follows:

Transfer state	Description
USBDTRST_BUSY	Transfer of current chunk is ongoing. This state is set by **usbd_receive()** or **usbd_send()** if the current transfer needs no processing by the base system. This can be difficult when performing a transfer on a double buffered OUT endpoint as there is a race condition between **usbd_receive()** and the interrupt handler. Both will update *usbd_transfer_t::state*. If the **usbd_receive()** changes the state to busy after the reception on the second buffer triggers the interrupt, but before this event is processed by the base system, the state freezes to busy.
USBDTRST_CHK	Chunk finished successfully and did not end with a short packet.
USBDTRST_SHORTPK	Chunk finished successfully and ended with a short packet.
USBDTRST_COMM	Chunk finished with an error.

2.3 Isochronous Endpoint Communication

Each isochronous endpoint has a FIFO attached to it by the class driver. The class driver has to call **usbd_fifo_attach()** from the class driver callback function. This function is executed before the handshake phase is performed for a *set configuration* or *set interface* request. Thus the host is not allowed to perform any transactions to the isochronous endpoint before the FIFO is attached.

The low level driver implements callback functions for the FIFO. OUT endpoints use the IN callback and IN endpoints use the OUT callback. The low level driver enables/disables event generation of the endpoint, based on its FIFO status.

2.4 Driver Configuration

In most cases a low level driver can work with multiple chips. Sometimes there are small differences between the supported chips that affect the low level driver. These include the number of endpoints, the maximum packet size for endpoints, and pull-up control. These parameters are captured in a header file. The main part of the name of this file ends with "cfg" or "config" (for example, **usbd_sam_config.h** or **usbd_sam_cfg.h**).

Sometimes the initialization of different chips needs to differ. Such differences are captured to functions in a **.c** file. Again the main part of the file name ends with "**config**" or "**cfg**." Usually function calls for initialize and delete suffice. These can handle GPIO or clock initialization differences.

The base system makes the following configuration parameters public:

Parameter	Description
USBD_SOFTMR_SUPPORT	If the value is non-zero, the application needs the SOF timer. The low level driver enables SOF interrupts and calls **softmr_tick()** when the interrupt strikes.
USBD_ISOCHRONOUS_SUPPORT	If the value is non-zero, the low level driver enables FIFO handling routines (the **usbd_add_fifo()** and **usbd_drop_fifo()** functions, FIFO callbacks, and FIFO-specific handing in the interrupt handler).
USBD_REMOTE_WAKEUP	If the value is non-zero, remote wakeup is supported by the device.

2.5 Endpoint Information Management

The base system is responsible for managing endpoint information. It manages a structure array called *usbd_ep_list*. Each item in the array holds information about an endpoint.

Each structure in the array has the following members:

Member	Description
*usbd_transfer_t *tr*	The transfer ongoing on the endpoint. This field is read-only for the low level driver.
*struct usbd_ep_info_s *next*	A pointer to the next endpoint of the same interface. This is only used by the base system.
*rngbuf_t *fifo*	Available only if the endpoint type is isochronous. The FIFO is attached to the endpoint. If no FIFO is attached the value is NULL. This field is read-only for the low level driver.
usbd_ep_handle_t eph	The handle for this endpoint. This is the "up-to-date" value changed by the **usbd_add_ep()** and **usbd_remove_ep()** functions in the base system. This is not used by the low level driver.
usbd_hw_ep_info_t hw_info	The layout of this structure is defined by the low level driver. This is a good place to store hardware-specific information related to an endpoint. For example, information about the double buffering state and the physical endpoint index can be stored here.
uint16_t psize	The packet size used for the endpoint. This field is read-only for the low level driver.
uint8_t ep_type	The endpoint type (see the EPD_ATTR_XXX values). This field is read-only for the low level driver.
uint8_t addr	The address of the endpoint. This field is read-only for the low level driver.
uint8_t age	The age of the endpoint. This is only used by the base system.
uint8_t halted	A flag used to record whether the endpoint is halted or not. This is only used by the base system.

3 API

This section documents the Application Programming Interface (API). It describes all the functions that must be provided by the low level driver for the EUSBD stack to operate correctly.

3.1 Module Management

usbd_hw_init

Use this function to initialize the low level driver and hardware for operation.

Typically this function performs clock initialization and resource allocation for the USB hardware interface. It does not enable operation of the driver.

> **Note**: The definition of the *usbd_hw_init_info_t* structure is target-specific and it may be a dummy structure. It is used to give systems flexibility in system configuration.

Format

```
int usbd_hw_init ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
USBD_SUCCESS	Successful operation.
USBD_ERROR	Operation failed.

usbd_hw_start

Use this function to start the low level driver.

This function typically activates the pull-up resistor and enables USB interrupts.

> **Note:** You must call **usbd_hw_init()** before this function.

Format

```
int usbd_hw_start ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
USBD_SUCCESS	Successful operation.
USBD_ERROR	Operation failed.

usbd_hw_stop

Use this function to stop the low level driver.

This function typically disables USB interrupts and disconnects the pull-up resistor to signal to the host that the USB device is not available. It reverses the operations performed by **usbd_hw_start()**.

Format

```
int usbd_hw_stop ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
USBD_SUCCESS	Successful operation.
USBD_ERROR	Operation failed.

usbd_hw_delete

Use this function to release resources allocated for the USB hardware interface.

This function can also include code to put the hardware into low power mode. It typically reverses the operations performed by **usbd_hw_init()**.

Format

```
int usbd_hw_delete ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
USBD_SUCCESS	Successful operation.
USBD_ERROR	Operation failed.

3.2 Driver Management

usbd_add_ep

Use this function to configure a new endpoint.

Properties of the endpoint (packet size, type, direction, and so on) can be found in the *usbd_ep_list* array, selected by index. This array has a structure of the type *usbd_hw_ep_info_t*. This type is defined by the low level driver in the file **usbd_dev.h**. This structure can be used to hold hardware-specific information on endpoints.

Format

```
int usbd_add_ep ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint to add.	int

Return Values

Return value	Description
USBD_SUCCESS	Operation successful.
USBD_ERROR	Operation failed.

usbd_remove_ep

Use this function to disable an endpoint.

This removes the specified endpoint from the *usbd_ep_list* array.

Format

```
void usbd_remove_ep ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint to disable.	int

Return Values

Return value
None.

usbd_send

Use this function to send data.

This function should set the transfer state (*tr->state*) to USBDTRST_BUSY.

This function initiates the requested transfer. After this it is the responsibility of the low level driver to send the requested data in one or more USB data packets, maintaining the transfer until it is completed or fails. The low level driver must:

- Read the data from the buffer passed to it (*tr->buffer*) and send it as it can. The total length of the data to send is given in *tr->length*.
- Maintain the counter of the number of bytes successfully transmitted in the transfer descriptor. This is *tr->csize*.
- If the transfer is not completed by the sending of a short packet, when the transfer is completed set the state of the transfer to USBDTRST_CHK and send an event.
- If the transfer is completed by the sending of a short packet, set the state of the transfer to USBDTRST_SHORTPK and send an event.

Format

```
int usbd_send ( usbd_transfer_t * tr )
```

Arguments

Argument	Description	Type
tr	Transfer descriptor holding detailed information about the transfer.	usbd_transfer_t *

Return Values

Return value	Description
USBDERR_NONE	Operation successful.
USBDERR_COMM	Communication error.
USBDERR_BUSY	Endpoint is busy.

usbd_receive

Use this function to receive data.

This function should set the transfer state (*tr->state*) to USBDTRST_BUSY. The transfer is completed when either the total number of bytes requested in the transfer has been received, or when a packet smaller than the USB packet size (a *short packet*) is received.

This function initiates the requested transfer and it is then the responsibility of the low level driver to receive the requested data in one or more USB data packets, maintaining the transfer until it is completed or fails. The driver must:

- Copy received data into the buffer passed to it (*tr->buffer*) as it receives it. The maximum length of the data to be received is given in *tr->length*.
- Maintain the counter of the number of bytes successfully received in the transfer descriptor. This is *tr->csize*.
- If the transfer is not completed by the reception of a short packet, when the transfer is completed set the state of the transfer to USBDTRST_CHK and send an event.
- If the transfer is completed by the reception of a short packet, set the state of the transfer to USBDTRST_SHORTPK and send an event.

Format

```
int usbd_receive ( usbd_transfer_t * tr )
```

Arguments

Argument	Description	Type
tr	Transfer descriptor holding detailed information about the transfer.	usbd_transfer_t *

Values

Return value	Description
USBDERR_NONE	Operation successful.
USBDERR_COMM	Communication error.
USBDERR_BUSY	Endpoint is busy.

usbd_add_fifo

Use this function to attach a FIFO to the specified isochronous endpoint.

> **Note:** This function is used only by isochronous endpoints.

When **usbd_add_ep()** is called, the FIFO is not yet available. The FIFO is attached later by the class driver. This function is called when the host configures the device using a "set configuration" or "set interface" request. The handshake for these requests is sent only after this call finishes; this ensures that the host cannot use the endpoint before the FIFO is attached.

This call sets callback routines for the FIFO. For OUT endpoints set the IN callback. For IN endpoints set the OUT callback.

The "fifo" member of the *usbd_ep_list* item is set by the base system.

Format

```
int usbd_add_fifo ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint to assign the FIFO to.	int

Return Values

Return value	Description
USBD_SUCCESS	Operation successful.
USBD_ERROR	Operation failed.

usbd_drop_fifo

Use this function to detach a FIFO from the specified isochronous endpoint.

> **Note:** This function is used only by isochronous endpoints.

The "fifo" member of the *usbd_ep_list* item is cleared by the base system.

Format

```
int usbd_drop_fifo ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint to drop the FIFO from.	int

Return Values

Return value	Description
USBD_SUCCESS	Operation successful.
USBD_ERROR	Operation failed.

usbd_at_high_speed

Use this function to find whether the hardware is communicating at high speed (480Mb/s).

Format

```
int usbd_at_high_speed ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
Zero	The device is not operating at high speed.
Non-zero	The device is operating at high speed.

usbd_get_stall

Use this function to find whether the specified endpoint is currently stalled.

An endpoint is said to be stalled if the USB device controller is sending stall handshakes in response to any request for that endpoint.

> **Note:** This function must be used for control, interrupt, and bulk endpoints.

Format

```
int usbd_get_stall ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint in *usbd_ep_list*.	int

Return Values

Return value	Description
Zero	The endpoint is not stalled.
Non-zero.	The endpoint is stalled.

usbd_set_stall

Use this function to set the specified endpoint to send stall handshakes.

> **Note:** This must be implemented for control, interrupt, and bulk endpoints.

The stall flag is provided by USB controllers on a per-endpoint basis. For control endpoints, the next setup packet must clear this flag.

Format

```
void usbd_set_stall ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint in *usbd_ep_list*.	int

Return Values

Return value
None.

usbd_clr_stall

Use this function to clear the stall flag for the specified endpoint.

> **Note:** This must be implemented for control, interrupt, and bulk endpoints.

The stall flag is provided by USB controllers on a per-endpoint basis. For control endpoints, the next setup packet must clear this flag.

Format

```
void usbd_clr_stall ( int index )
```

Arguments

Argument	Description	Type
index	The index of the endpoint in *usbd_ep_list*.	int

Return Values

Return value
None.

usbd_get_state

Use this function to get the current driver state.

Format

```
int usbd_get_state ( void )
```

Arguments

Argument
None.

Return Values

Return value
A USBDST_XXX value.

usbd_set_addr_post

This function is called by the base system when the device receives a "set address" request.

The call is made after the handshake has been sent back to the host. If the hardware module can only change the address after the handshake is sent, use this function to change the device address.

Format

```
void usbd_set_addr_post ( uint8_t daddr )
```

Arguments

Argument	Description	Type
daddr	The new device address.	uint8_t

Return Values

Return value
None.

usbd_set_addr_pre

This function is called by the base system when the device receives a "set address" request.

This function is called immediately after the setup packet has been processed and before the handshake is sent back. If the target hardware module can only change the address after the handshake is sent, use this function to set the device address.

Format

```
void usbd_set_addr_pre ( uint8_t daddr )
```

Arguments

Argument	Description	Type
daddr	The new device address.	uint8_t

Return Values

Return value
None.

usbd_set_cfg_post

This function is called by the base system when the device receives a "set address" request.

The call is made after the handshake has been sent back to the host. If the hardware module can only change the address after the handshake is sent, use this function to change the device address.

Format

```
void usbd_set_cfg_post ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

usbd_set_cfg_pre

This function is called by the base system when the device receives a "set address" request.

The call is made after the handshake has been sent back to the host. If the hardware module can only change the address after the handshake is sent, use this function to change the device address.

Format

```
void usbd_set_cfg_pre ( void )
```

Arguments

Argument
None.

Return Values

Return value
None.

usbd_set_testmode

Use this function to specify the test mode.

> **Note:** This is only useful if USBD_TEST_MODE_SUPPORT is enabled in **src/config/config_usbd.h**.

Format

```
void usbd_set_testmode ( uint8_t testmode_selector )
```

Arguments

Argument	Description	Type
testmode_selector	The test mode.	uint8_t

Return Values

Return value
None.

3.3 Callback Functions

This section describes the callback functions that the driver uses to indicate specified events to the USBD stack.

> **Note:** It is the user's responsibility to provide these functions.

usbd_pup_on_off

The low level driver calls this callback function to enable or disable the pull-up resistor.

This function is needed only if the USB hardware has no on-chip pull-up resistor. It is called by the low level driver and implemented by the application.

Format

```
int usbd_pup_on_off ( int on )
```

Arguments

Argument	Description	Type
on	One of the following: • Non-zero to activate the pull-up resistor. • Zero to deactivate it.	int

Return Values

Return value	Description
Zero	Successful execution.
Non-zero	Operation failed.

usbd_is_self_powered

The low level driver calls this callback function to find whether the device is currently running self-powered.

If a device is not self-powered, it is powered from the USB.

Format

```
int usbd_is_self_powered ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
Zero	The device is bus-powered.
1	The device is self-powered.

usbd_conn_state

The low level driver calls this callback function to find the current device state.

Format

```
void usbd_conn_state ( usbd_conn_state_t new_state )
```

Arguments

Argument	Description	Type
new_state	One of the following: • Non-zero to activate self-powered mode. • Zero to activate bus-powered mode.	usbd_conn_state_t

Return Values

Return value
None.

usbd_vbus

This callback function reports the current V_{BUS} state to the USB stack.

It is a requirement of USB to remove the pull-up resistor when V_{BUS} is off (and the device is connected to the host), so the system needs to tell the stack when to complete the required actions. If the low level driver has no V_{BUS} handling, call this function when the V_{BUS} is detected or removed. Otherwise this function is not needed.

Format

```
void usbd_vbus (
    int    on,
    int    in_irq_context )
```

Arguments

Argument	Description	Type
on	One of the following: • Non-zero when V_{BUS} is detected. • Zero when V_{BUS} is removed.	int
in_irq_context	Set this to 1 if this is called from an interrupt.	int

Return Values

Return value
None.

3.4 Error Codes

If a function executes successfully it returns with USBD_SUCCESS, a value of zero.

Argument	Description	Type
USBD_SUCCESS	0	Operation successful.
USBDERR_NONE	0	No error.
USBD_ERROR	1	Operation failed.
USBDERR_BUSY	1	Endpoint is busy.
USBDERR_INVALIDEP	3	Invalid endpoint.
USBDERR_NOTREADY	5	Transfer cannot be started.
USBDERR_INTERNAL	6	Internal error.
USBDERR_COMM	7	Communication error.

3.5 Types and Definitions

usbd_iso_buffer_t

This structure is only used if USBD_ISOCHRONOUS_SUPPORT is enabled.

Parameter name	Description	Type
* data	Pointer to the data.	void
size	Size of the buffer.	uint32_t
nbytes	Number of bytes.	uint32_t
rd_ndx		uint32_t

USB Driver State Values

The driver may be in any of the following states:

Value	Description
USBDST_DISABLED	State after **usbd_init()**.
USBDST_DEFAULT	State after USB reset.
USBDST_ADDRESSED	State after "set address" request completed.
USBDST_CONFIGURED	State after "set configuration" request completed.
USBDST_SUSPENDED	State after suspend.

USB Transfer State Values

A transfer may be in any of the following states:

Value	Description
USBDTRST_DONE	Transfer ended.
USBDTRST_BUSY	Low level driver is busy.
USBDTRST_CHK	Check status.
USBDTRST_SHORTPK	Ended with short packet.
USBDTRST_EP_KILLED	Failed; endpoint no longer available.
USBDTRST_COMM	Communication error.

Test Modes

There are four test modes.

Value	Description
USBD_TEST_J_MODE	J mode.
USBD_TEST_K_MODE	K mode.
USBD_TEST_SE0_NAK	Single Ended Zero NAK.
USBD_TEST_PACKET_MODE	Packet mode.

Events

usbd_stprx_event

This event is sent by the low level driver when a setup packet is received successfully.

usbd_bus_event

This event is sent by the low level driver when an event such as a reset, suspend, or wakeup occurs on the bus.

Endpoint Attributes

These are defined in the base package file **usbd_std.h**:

Value	Description
EPD_ATTR_CTRL	Control endpoint.
EPD_ATTR_ISO	Isochronous endpoint.
EPD_ATTR_BULK	Bulk endpoint.
EPD_ATTR_INT	Interrupt endpoint.

4 Integration

4.1 OS Abstraction Layer (OAL)

All HCC modules use the OS Abstraction Layer (OAL) that allows the module to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The low level driver uses the following OAL components:

OAL Resource	Number Required
Tasks	2
Mutexes	1
Events	3

4.2 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

The driver makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.

The driver makes use of the following standard PSP macro:

Macro	Package	Element	Description
PSP_WR_LE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as little-endian to a memory location.