

 **μC/USB Host™**  
*Universal Serial Bus Host Stack*

**User's Manual**  
**V3.41**

**Micrium®**  
For the Way Engineers Work

Micrium

1290 Weston Road, Suite 306

Weston, FL 33326

USA

[www.micrium.com](http://www.micrium.com)

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where Micrium Press is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 2013 by Micrium except where noted otherwise. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs and code examples in this book are presented for instructional value. The programs and examples have been carefully tested, but are not guaranteed to any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher assumes no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

# Table of Contents

<b>Chapter 1</b>	About USB .....	8
<b>1-1</b>	Introduction .....	8
<b>1-1-1</b>	Bus Topology .....	8
<b>1-1-2</b>	USB Host .....	9
<b>1-1-3</b>	USB Device .....	9
<b>1-2</b>	Data Flow Model .....	10
<b>1-2-1</b>	Endpoint .....	10
<b>1-2-2</b>	Pipes .....	11
<b>1-2-3</b>	Transfer Types .....	11
<b>1-3</b>	Physical Interface and Power Management .....	14
<b>1-3-1</b>	Speed .....	14
<b>1-3-2</b>	Power Distribution .....	15
<b>1-4</b>	Device Structure and Enumeration .....	15
<b>1-4-1</b>	USB Device Structure .....	15
<b>1-4-2</b>	Device States .....	17
<b>1-4-3</b>	Enumeration .....	18
<b>Chapter 2</b>	Getting Started .....	20
<b>2-1</b>	Prerequisites .....	21
<b>2-2</b>	Downloading the Source Code Files .....	22
<b>2-3</b>	Installing the files .....	23
<b>2-4</b>	Building the Sample Application .....	25
<b>2-4-1</b>	Understanding Micrium Examples .....	25
<b>2-4-2</b>	Including USB Host Stack Source Code .....	27
<b>2-4-3</b>	Copying and Modifying Template Files .....	28
<b>2-4-4</b>	Modifying the Application Configuration File .....	32
<b>2-5</b>	Running the Sample Application .....	33

---

<b>Chapter 3</b>	Architecture .....	38
<b>3-1</b>	USB Host Stack Overview .....	39
<b>3-1-1</b>	USB Host Stack Modules .....	40
<b>3-1-2</b>	USB Host Stack Dependencies .....	43
<b>3-2</b>	Sending and Receiving Data .....	44
<b>3-3</b>	Task Model .....	44
<b>3-3-1</b>	Hub Task .....	44
<b>3-3-2</b>	Asynchronous Task .....	46
<b>3-4</b>	Enumeration Process .....	46
<b>Chapter 4</b>	Configuration .....	48
<b>4-1</b>	Static Stack Configuration .....	48
<b>4-1-1</b>	USB Host Configuration .....	49
<b>4-1-2</b>	USB Classes Configuration .....	52
<b>4-1-3</b>	Debug Configuration .....	53
<b>4-2</b>	Application Specific Configuration .....	54
<b>4-2-1</b>	Task Priorities .....	55
<b>4-2-2</b>	Task Stack Sizes .....	55
<b>4-3</b>	Host Controller Driver Configuration .....	56
<b>4-3-1</b>	Host Controller Configuration Structure .....	56
<b>4-3-2</b>	Host Controller Initialization .....	58
<b>4-4</b>	Configuration Examples .....	62
<b>4-4-1</b>	Single Host Controller and Unique Device .....	63
<b>4-4-2</b>	Single Host Controller and Multiple Devices .....	64
<b>4-4-3</b>	Multi-Host Controllers and Multiple Devices .....	67
<b>Chapter 5</b>	Host Driver Guide .....	72
<b>5-1</b>	Host Driver Model .....	73
<b>5-2</b>	Host Driver API .....	73
<b>5-3</b>	Interrupt Handling .....	77
<b>5-3-1</b>	Single USB ISR Vector with ISR Handler Argument .....	78
<b>5-3-2</b>	Single USB ISR Vector .....	78
<b>5-3-3</b>	Multiple USB ISR Vectors with ISR Handler Arguments .....	79
<b>5-3-4</b>	Multiple USB ISR Vectors .....	79
<b>5-4</b>	Host Controller Driver Configuration .....	80
<b>5-5</b>	Memory Allocation .....	80

---

<b>5-6</b>	CPU and Board Support .....	81
<b>5-7</b>	USB Host Controller Driver Functional Model .....	82
<b>5-7-1</b>	Root Hub Interactions .....	82
<b>5-7-2</b>	Endpoint Opening .....	83
<b>5-7-3</b>	URB Submit .....	84
<b>Chapter 6</b>	Communication Device Class .....	90
<b>6-1</b>	Overview .....	91
<b>6-2</b>	Class Implementation .....	94
<b>6-3</b>	Configuration and Initialization .....	95
<b>6-3-1</b>	General Configuration .....	95
<b>6-3-2</b>	Class Initialization .....	95
<b>6-3-3</b>	Device Connection and Disconnection Handling .....	96
<b>6-4</b>	Abstract Control Model (ACM) Subclass .....	98
<b>6-4-1</b>	Configuration and Initialization .....	98
<b>6-4-2</b>	Connection and Disconnection Handling .....	99
<b>6-4-3</b>	Demo Application .....	101
<b>Chapter 7</b>	Future Technology Devices International Class .....	104
<b>7-1</b>	Overview .....	105
<b>7-1-1</b>	Endpoints .....	105
<b>7-1-2</b>	Class Requests .....	106
<b>7-1-3</b>	Serial status .....	107
<b>7-2</b>	Class Implementation .....	107
<b>7-3</b>	Configuration and Initialization .....	108
<b>7-3-1</b>	General Configuration .....	108
<b>7-3-2</b>	Class Initialization .....	108
<b>7-4</b>	Demo Application .....	109
<b>7-5</b>	Demo Application Configuration .....	112
<b>Chapter 8</b>	Human Interface Device Class .....	114
<b>8-1</b>	Overview .....	115
<b>8-1-1</b>	Report .....	115
<b>8-2</b>	Class Implementation .....	120
<b>8-3</b>	Configuration and Initialization .....	121
<b>8-3-1</b>	General Configuration .....	121
<b>8-3-2</b>	Class Initialization .....	122

---

<b>8-3-3</b>	Device Connection and Disconnection Handling .....	123
<b>8-4</b>	Demo Application .....	125
<b>8-4-1</b>	Demo Application Configuration .....	126
<b>Chapter 9</b>	Mass Storage Class .....	127
<b>9-1</b>	Overview .....	128
<b>9-1-1</b>	Mass Storage Class Protocol .....	128
<b>9-1-2</b>	Endpoints .....	129
<b>9-1-3</b>	Mass Storage Class Requests .....	129
<b>9-1-4</b>	Small Computer System Interface (SCSI) .....	130
<b>9-2</b>	Class Implementation .....	130
<b>9-3</b>	Configuration and Initialization .....	132
<b>9-3-1</b>	General Configuration .....	132
<b>9-3-2</b>	Class Initialization .....	132
<b>9-3-3</b>	Device Connection and Disconnection Handling .....	133
<b>9-4</b>	Demo Application .....	134
<b>9-4-1</b>	Demo Application Configuration .....	137
<b>Chapter 10</b>	Porting $\mu$ C/USB-Host to your Kernel .....	139
<b>10-1</b>	Overview .....	139
<b>10-2</b>	Porting the Stack to your Kernel .....	140
<b>10-2-1</b>	Task Creation .....	140
<b>10-2-2</b>	Semaphore .....	141
<b>10-2-3</b>	Mutex .....	142
<b>10-2-4</b>	Message Queue .....	143
<b>Appendix A</b>	Core API Reference .....	145
<b>A-1</b>	Host Functions .....	146
<b>A-2</b>	Host Controller Functions .....	151
<b>A-3</b>	Class Management Functions .....	158
<b>A-4</b>	Kernel Abstraction Functions .....	160
<b>Appendix B</b>	CDC API Reference .....	179
<b>B-1</b>	CDC Functions .....	180
<b>B-2</b>	ACM Functions .....	184

---

<b>Appendix C</b>	FTDI API Reference .....	205
<b>C-1</b>	FTDI Functions .....	206
<b>Appendix D</b>	HID API Reference .....	231
<b>D-1</b>	HID Functions .....	232
<b>Appendix E</b>	MSC API Reference .....	253
<b>E-1</b>	MSC Functions .....	254
<b>E-2</b>	File System MSC Driver Functions .....	256
<b>Appendix F</b>	Host Controller Driver API Reference .....	259
<b>F-1</b>	Host Driver Functions .....	260
<b>F-2</b>	Root Hub Driver Functions .....	278
<b>F-3</b>	Host Driver BSP Functions .....	292
<b>Appendix G</b>	Error Codes .....	295
<b>G-1</b>	Generic Error Codes .....	296
<b>G-2</b>	Device Error Codes .....	296
<b>G-3</b>	Configuration Error Codes .....	296
<b>G-4</b>	Interface Error Codes .....	297
<b>G-5</b>	Endpoint Error Codes .....	297
<b>G-6</b>	URB Error Codes .....	297
<b>G-7</b>	Descriptor Error Codes .....	298
<b>G-8</b>	Host Controller Error Codes .....	298
<b>G-9</b>	Kernel Layer Error Codes .....	298
<b>G-10</b>	Class Error Codes .....	299
<b>G-11</b>	HUB Class Error Codes .....	299
<b>G-12</b>	Human Interface Device (HID) Class Error Codes .....	299
<b>G-13</b>	Mass Storage Class (MSC) Error Codes .....	300
<b>G-14</b>	future technology devices international (ftdi) class Error Codes .....	300

# Chapter

# 1

## About USB

This chapter presents a quick introduction to USB. The first section in this chapter introduces the basic concepts of the USB specification Revision 2.0. The second section explores the data flow model. The third section gives details about the device operation. Lastly, the fourth section describes USB device logical organization.

The full protocol is described extensively in the USB Specification Revision 2.0 at <http://www.usb.org>.

### **1-1 INTRODUCTION**

The Universal Serial Bus (USB) is an industry standard maintained by the USB Implementers Forum (USB-IF) for serial bus communication. The USB specification contains all the information about the protocol such as the electrical signaling, the physical dimension of the connector, the protocol layer, and other important aspects. USB provides several benefits compared to other communication interfaces such as ease of use, low cost, low power consumption and, fast and reliable data transfer.

#### **1-1-1 BUS TOPOLOGY**

USB can connect a series of devices using a tiered star topology. The key elements in USB topology are the *host*, *hubs*, and *devices*, as illustrated in Figure 1-1. Each node in the illustration represents a USB hub or a USB device. At the top level of the graph is the root hub, which is part of the host. There is only one host in the system. The specification allows up to seven tiers and a maximum of five non-root hubs in any path between the host and a device. Each tier must contain at least one hub except for the last tier where only devices are present. Each USB device in the system has a unique address assigned by the host through a process called *enumeration* (see section 1-4-3 on page 18 for more details on enumeration).



The host learns about the device capabilities during enumeration, which allows the host operating system to load a specific driver for a particular USB device. The maximum number of peripherals that can be attached to a host is 127, including the root hub.

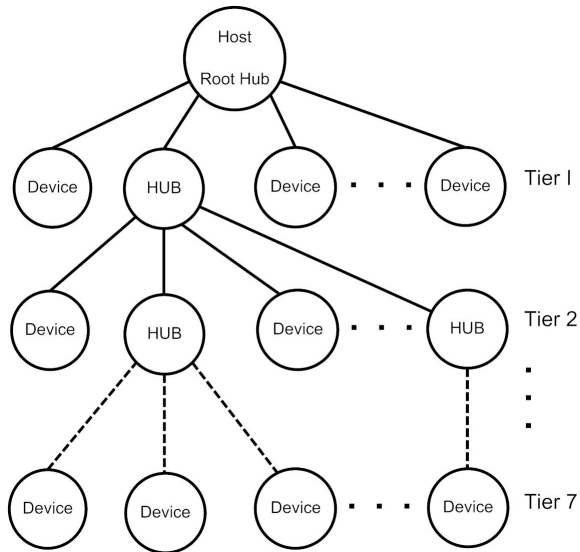


Figure 1-1 **Bus topology**

### **1-1-2 USB HOST**

The USB host communicates with the devices using a USB host controller. The host is responsible for detecting and enumerating devices, managing bus access, performing error checking, providing and managing power, and exchanging data with the devices.

### **1-1-3 USB DEVICE**

A USB device implements one or more USB *functions* where a function provides one specific capability to the system. Examples of USB functions are keyboards, webcam, speakers, or a mouse. The requirements of the USB functions are described in the USB class specification. For example, keyboards and mice are implemented using the Human Interface Device (HID) specification.

USB devices must also respond to requests from the host. For example, on power up, or when a device is connected to the host, the host queries the device capabilities during enumeration, using standard requests.

## **1-2 DATA FLOW MODEL**

This section defines the elements involved in the transmission of data across USB.

### **1-2-1 ENDPOINT**

*Endpoints* function as the point of origin or the point of reception for data. An endpoint is a logical entity identified using an endpoint address. The endpoint address of a device is fixed, and is assigned when the device is designed, as opposed to the device address, which is assigned by the host dynamically during enumeration. An endpoint address consists of an endpoint number field (0 to 15), and a direction bit that indicates if the endpoint sends data to the host (IN) or receives data from the host (OUT). The maximum number of endpoints allowed on a single device is 32.

Endpoints contain configurable characteristics that define the behavior of a USB device:

- Bus access requirements
- Bandwidth requirement
- Error handling
- Maximum packet size that the endpoint is able to send or receive
- Transfer type
- Direction in which data is sent and receive from the host

### **ENDPOINT ZERO REQUIREMENT**

Endpoint zero (also known as Default Endpoint) is a bi-directional endpoint used by the USB host system to get information, and configure the device via standard requests. All devices must implement an endpoint zero configured for control transfers (see section “Control Transfers” on page 11 for more information).

## **1-2-2 PIPES**

A USB pipe is a logical association between an endpoint and a software structure in the USB host software system. USB pipes are used to send data from the host software to the device's endpoints. A USB pipe is associated to a unique endpoint address, type of transfer, maximum packet size, and interval for transfers.

The USB specification defines two types of pipes based on the communication mode:

- Stream Pipes: Data carried over the pipe is unstructured.
- Message Pipes: Data carried over the pipe has a defined structure.

The USB specification requires a default control pipe for each device. A default control pipe uses endpoint zero. The default control pipe is a bi-directional message pipe.

## **1-2-3 TRANSFER TYPES**

The USB specification defines four transfer types that match the bandwidth and services requirements of the host and the device application using a specific pipe. Each USB transfer encompasses one or more transactions that send data to and from the endpoint. The notion of transactions is related to the maximum payload size defined by each endpoint type, that is when a transfer is greater than this maximum, it will be split into one or more transactions to fulfill the action.

### **CONTROL TRANSFERS**

Control transfers are used to configure and retrieve information about the device capabilities. They are used by the host to send standard requests during and after enumeration. Standard requests allow the host to learn about the device capabilities; for example, how many and which functions the device contains. Control transfers are also used for class-specific and vendor-specific requests.

A control transfer contains three stages: Setup, Data, and Status. These stages are listed in Table 1-1.

Stage	Description
Setup	The Setup stage includes information about the request. This SETUP stage represents one transaction.
Data	The Data stage contains data associated with request. Some standard and class-specific request may not require a Data stage. This stage is an IN or OUT directional transfer and the complete Data stage represents one or more transactions.
Status	The Status stage, representing one transaction, is used to report the success or failure of the transfer. The direction of the Status stage is opposite to the direction of the Data stage. If the control transfer has no Data stage, the Status stage always is from the device (IN).

Table 1-1 **Control Transfer Stages**

## **BULK TRANSFERS**

Bulk transfers are intended for devices that exchange large amounts of data where the transfer can take all of the available bus bandwidth. Bulk transfers are reliable, as error detection and retransmission mechanisms are implemented in hardware to guarantee data integrity. However, bulk transfers offer no guarantee on timing. Printers and mass storage devices are examples of devices that use bulk transfers.

## **INTERRUPT TRANSFERS**

Interrupt transfers are designed to support devices with latency constraints. Devices using interrupt transfers can schedule data at any time. Devices using interrupt transfer provide a polling interval which determines when the scheduled data is transferred over the bus. Interrupt transfers are typically used for event notifications.

## **ISOCRONOUS TRANSFERS**

Isochronous transfers are used by devices that require data delivery at a constant rate with a certain degree of error-tolerance. Retransmission is not supported by isochronous transfers. Audio and video devices use isochronous transfers.

## **USB DATA FLOW MODEL**

Table 1-2 shows a graphical representation of the data flow model.

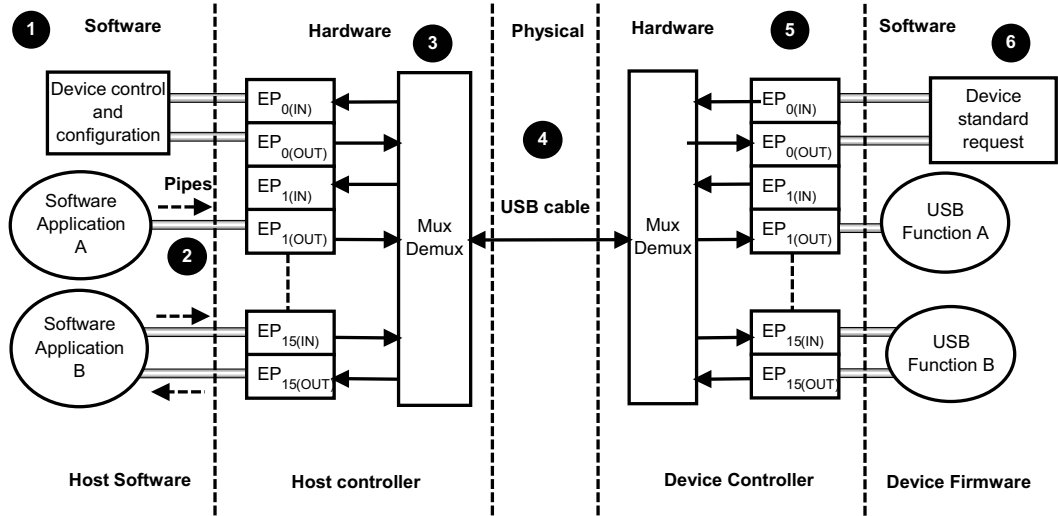


Figure 1-2 **USB data flow**

- F1-2(1) The host software uses standard requests to query and configure the device using the default pipe. The default pipe uses endpoint zero (EP0).
- F1-2(2) USB pipes allow associations between the host application and the device's endpoints. Host applications send and receive data through USB pipes.
- F1-2(3) The host controller is responsible for the transmission, reception, packing and unpacking of data over the bus.
- F1-2(4) Data is transmitted via the physical media.
- F1-2(5) The device controller is responsible for the transmission, reception, packing and unpacking of data over the bus. The USB controller informs the USB device software layer about several events such as bus events and transfer events.
- F1-2(6) The device software layer responds to the standard request, and implements one or more USB functions as specified in the USB class document.

## TRANSFER COMPLETION

The notion of transfer completion is only relevant for control, bulk and interrupt transfers as isochronous transfers occur continuously and periodically by nature. In general, control, bulk and interrupt endpoints must transmit data payload sizes that are less than or equal to the endpoint's maximum data payload size. When a transfer's data payload is greater than the maximum data payload size, the transfer is split into several transactions whose payload is maximum-sized except the last transaction which contains the remaining data. A transfer is deemed complete when:

- The endpoint transfers exactly the amount of data expected.
- The endpoint transfers a short packet, that is a packet with a payload size less than the maximum.
- The endpoint transfers a zero-length packet.

## 1-3 PHYSICAL INTERFACE AND POWER MANAGEMENT

USB transfers data and provides power using four-wire cables. The four wires are:  $V_{\text{bus}}$ ,  $D^+$ ,  $D^-$  and Ground. Signaling occurs on the  $D^+$  and  $D^-$  wires.

### 1-3-1 SPEED

The USB 2.0 specification defines three different speeds.

- Low Speed: 1.5 Mb/s
- Full Speed: 12 Mb/s
- High Speed: 480 Mb/s

## **1-3-2 POWER DISTRIBUTION**

The host can supply power to USB devices that are directly connected to the host. USB devices may also have their own power supplies. USB devices that use power from the cable are called bus-powered devices. Bus-powered devices can draw a maximum of 500 mA from the host. USB devices that have an alternative source of power are called self-powered devices.

## **1-4 DEVICE STRUCTURE AND ENUMERATION**

Before the host application can communicate with a device, the host needs to understand the capabilities of the device. This process takes place during device enumeration. After enumeration, the host can assign and load a specific driver to allow communication between the application and the device.

During enumeration, the host assigns an address to the device, reads descriptors from the device, and selects a configuration that specifies power and interface requirements. In order for the host to learn about the device's capabilities, the device must provide information about itself in the form of descriptors.

This section describes the device's logical organization from the USB host's point of view.

### **1-4-1 USB DEVICE STRUCTURE**

From the host's point of view, USB devices are internally organized as a collection of configurations, interfaces and endpoints.

#### **CONFIGURATION**

A USB configuration specifies the capabilities of a device. A configuration consists of a collection of USB interfaces that implement one or more USB functions. Typically only one configuration is required for a given device. However, the USB specification allows up to 255 different configurations. During enumeration, the host selects a configuration. Only one configuration can be active at a time. The device uses a *configuration descriptor* to inform the host about a specific configuration's capabilities.

## **INTERFACE**

A USB interface or a group of interfaces provides information about a function or class implemented by the device. An interface can contain multiple mutually exclusive settings called *alternate settings*. The device uses an *interface descriptor* to inform the host about a specific interface's capabilities. Each interface descriptor contains a class, subclass, and protocol codes defined by the USB-IF, and the number of endpoints required for a particular class implementation.

## **ALTERNATE SETTINGS**

Alternate settings are used by the device to specify mutually exclusive settings for each interface. The default alternate settings contain the default settings of the device. The device also uses an interface descriptor to inform the host about an interface's alternate settings.

## **ENDPOINT**

An interface requires a set of endpoints to communicate with the host. Each interface has different requirements in terms of the number of endpoints, transfer type, direction, maximum packet size, and maximum polling interval. The device sends an endpoint descriptor to notify the host about endpoint capabilities.

Figure 1-3 shows the hierarchical organization of a USB device. Configurations are grouped based on the device's speed. A high-speed device might have a particular configuration in both high-speed and low/full speed.



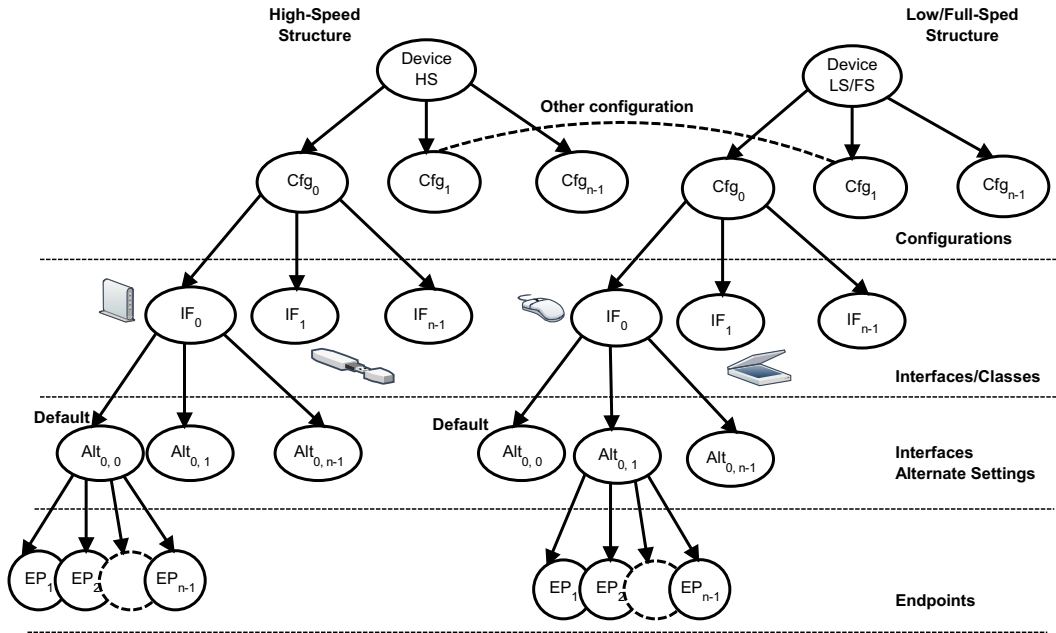


Figure 1-3 USB device structure

### 1-4-2 DEVICE STATES

The USB 2.0 specification defines six different states and are listed in Table 1-2.

Device States	Description
Attached	The device is in the Attached state when it is connected to the host or a hub port. The hub must be connected to the host or to another hub.
Powered	A device is considered in the Powered state when it starts consuming power from the bus. Only bus-powered devices use power from the host. Self-powered devices are in the Powered state after port attachment.
Default	After the device has been powered, it should not respond to any request or transactions until it receives a reset signal from the host. The device enters in the Default state when it receives a reset signal from the host. In the Default state, the device responds to standard requests at the default address 0.
Address	During enumeration, the host assigns a unique address to the device. When this occurs, the device moves from the Default state to the Address state.

---

<b>Device States</b>	<b>Description</b>
Configured	After the host assigns an address to the device, the host must select a configuration. After the host selects a configuration, the device enters the Configured state. In this state, the device is ready to communicate with the host applications.
Suspended	The device enters into Suspended state when no traffic has been seen over the bus for a specific period of time. The device retains the address assigned by the host in the Suspended state. The device returns to the previous state after traffic is present in the bus.

Table 1-2 **USB Device States**

### **1-4-3 ENUMERATION**

Enumeration is the process where the host configures the device and learns about the device's capabilities. The host starts enumeration after the device is attached to one of the root or external hub ports. The host learns about the device's manufacturer, vendor/product IDs and release versions by sending a *Get Descriptor* request to obtain the device descriptor and the maximum packet size of the default pipe (control endpoint 0). Once that is done, the host assigns a unique address to the device which will tell the device to only answer requests at this unique address. Next, the host gets the capabilities of the device by a series of *Get Descriptor* requests. The host iterates through all the available configurations to retrieve information about number of interfaces in each configuration, interfaces classes, and endpoint parameters for each interface and will lastly finish the enumeration process by selecting the most suitable configuration.



## Chapter

# 2

## Getting Started

This chapter gives you some insight into how to install and use the  $\mu$ C/USB-Host stack. The following topics are explained in this chapter:

- Prerequisites
- Downloading the source code files
- Installing the files
- Building the sample application
- Running the sample application

After the completion of this chapter, you should be able to build and run your first USB host application using the  $\mu$ C/USB-Host stack.

## 2-1 PREREQUISITES

Before running your first USB host application, you must ensure that you have a minimal set of required tools and components:

- A toolchain/IDE for your specific microcontroller
- A development board
- The  $\mu$ C/USB-Host stack with at least one of its USB class
- A Host Controller Driver (HCD) that is compatible with your hardware for the  $\mu$ C/USB-Host stack
- A Board Support Package (BSP) for your development board
- Other software products required by  $\mu$ C/USB-Host ( $\mu$ C/OS-II or  $\mu$ C/OS-III,  $\mu$ C/CPU,  $\mu$ C/LIB, and  $\mu$ C/FS, when the Mass Storage Class is used)
- An example project

If Micrium does not support your USB host controller or BSP, you will have to write your own HCD. Refer to Chapter 5, “Host Driver Guide” on page 72 for more information on writing your own USB HCD. You can also ask Micrium to develop the driver for your specific host controller.

## 2-2 DOWNLOADING THE SOURCE CODE FILES

μC/USB-Host can be downloaded from the Micrium customer portal. The distribution package includes the full source code and documentation. You can log into the Micrium customer portal at the address below to begin your download (you must have a valid license to gain access to the file):

<http://micrium.com/login>

μC/USB-Host depends on other modules, and you need to install all the required modules before building your application. Depending on the availability of support for your hardware platform, ports and drivers may or may not be available for download from the customer portal. Table 2-1 shows the module dependency for μC/USB-Host.

Module Name	Required	Note(s)
μC/USB-Host Core	YES	Hardware independent USB Host core stack.
μC/USB-Host Driver	YES	USB Host Controller Driver (HCD). Available only if Micrium supports your controller, otherwise you have to develop it yourself or ask Micrium to do it for you. Certain multi-host configurations might require to have more than one driver.
μC/USB-Host HID Class	Optional	Available only if you purchased the Human Interface Device (HID) class.
μC/USB-Host MSC Class	Optional	Available only if you purchased the Mass Storage Class (MSC).
μC/USB-Host CDC ACM Class	Optional	Available only if you purchased the Communication Device Class (CDC) with the Abstract Control Model (ACM) subclass.
μC/USB-Host FTDI Class	Optional	Available only if you purchased the Future Technology Devices International (FTDI) class.
μC/CPU Core	YES	Provides CPU specific data types and functions.
μC/CPU Port	YES	Available only if Micrium has support for your target architecture
μC/LIB Core	YES	Micrium run-time library
μC/LIB Port	Optional	Available only if Micrium has support for your target architecture
μC/OS-II Core	Optional	Available only if your application is using μC/OS-II
μC/OS-II Port	Optional	Available only if Micrium has support for your target architecture
μC/OS-III Core	Optional	Available only if your application is using μC/OS-III
μC/OS-III Port	Optional	Available only if Micrium has support for your target architecture

Module Name	Required	Note(s)
μC/FS	Optional	Micrium's File System (FS). Required only if your application uses the Mass Storage Class.
μC/FS Driver for MSC device	Optional	Required only if your application uses the Mass Storage Class.
μC/Clk	Optional	Required for μC/FS

Table 2-1 μC/USB-Host Module Dependency

Table 2-1 indicates that all the μC/USB-Host classes are optional because there is no required class to purchase with the μC/USB-Host Core and Driver. The class to purchase will depend on your needs. But don't forget that you need a class to build a complete USB project. Table 2-1 also indicates that μC/OS-II and -III Core and Port are optional. Indeed, μC/USB-Host stack does not assume a specific kernel to work with, but it still requires one.

### 2-3 INSTALLING THE FILES

Once all the distribution packages have been downloaded to your host machine, extract all the files at the root of your C:\ drive for instance. The package may be extracted to any location. After extracting all the files, the directory structure should look as illustrated in Figure 2-1. In the example, all Micrium products sub-folders shown in Figure 2-1 will be located in C:\Micrium\Software\. Note that the μC/FS and μC/Clk products are shown in Figure 2-1 but are only required if the Mass Storage Class is used.

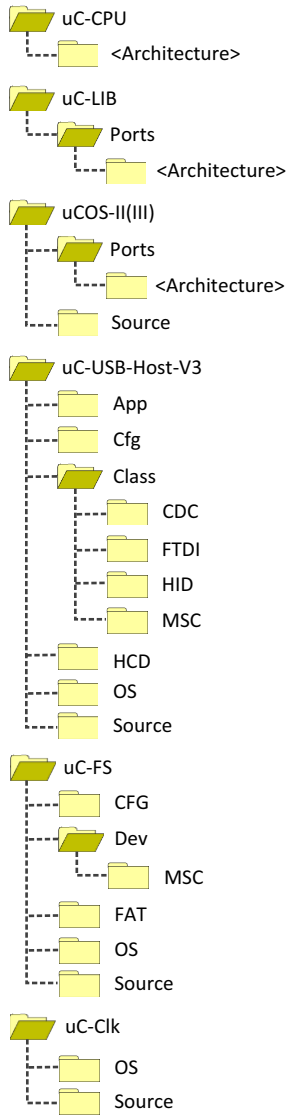


Figure 2-1 **Directory Tree for μC/USB-Host**



## 2-4 BUILDING THE SAMPLE APPLICATION

This section describes all the steps required to build a USB-based application. The instructions provided in this section are not intended for any particular toolchain, but instead are described in a generic way that can be adapted to any toolchain.

The best way to start building a USB-based project is to start from an existing project. If you are using  $\mu\text{C}/\text{OS-II}$  or  $\mu\text{C}/\text{OS-III}$ , Micrium provides example projects for multiple development boards and compilers. If your target board is not listed on Micrium's web site, you can download an example project for a similar board or microcontroller.

The purpose of the sample project is to allow your USB host to enumerate a device and perform basic communication with HID, MSC, FTDI or CDC ACM devices depending on which class(es) you purchased. After you have successfully completed and run the sample project, you can use it as a starting point to run other USB class demos you may have purchased.

$\mu\text{C}/\text{USB-Host}$  requires a Real-Time Operating System (RTOS). The following assumes that you have a working example project running on  $\mu\text{C}/\text{OS-II}$  or  $\mu\text{C}/\text{OS-III}$ .

### 2-4-1 UNDERSTANDING MICRIUM EXAMPLES

A Micrium example project is usually placed in the following directory structure.

```
\Micrium
  \Software
    \EvalBoards
      \<manufacturer>
        \<board_name>
          \<compiler>
            \<project name>
              \*.*
```

Note that Micrium does *not* provide by default an example project with the  $\mu\text{C}/\text{USB-Host}$  distribution package. Micrium examples are provided to customers in specific situations. If it happens that you receive a Micrium example, the directory structure shown above is generally used by Micrium. You may use a different directory structure to store the application and toolchain projects files.

### **\Micrium**

This is where Micrium places all software components and projects. This directory is generally located at the root directory.

### **\Software**

This sub-directory contains all software components and projects.

### **\EvalBoards**

This sub-directory contains all projects related to evaluation boards supported by Micrium.

### **\<manufacturer>**

This is the name of the manufacturer of the evaluation board. In some cases this can also be the name of the microcontroller manufacturer.

### **\<board name>**

This is the name of the evaluation board.

### **\<compiler>**

This is the name of the compiler or compiler manufacturer used to build the code for the evaluation board.

### **\<project name>**

The name of the project that will be demonstrated. For example a simple  $\mu$ C/USB-Host with  $\mu$ C/OS-III project might have the project name 'uCOS-III-USBH'.

### **\\*.\***

These are the source files for the project. This directory contains configuration files **app\_cfg.h**, **os\_cfg.h**, **os\_cfg\_app.h**, **cpu\_cfg.h** and other project-required sources files.

**os\_cfg.h** is a configuration file used to configure  $\mu$ C/OS parameters such as the maximum number of tasks, events, objects, which  $\mu$ C/OS services are enabled (semaphores, mailboxes, queues), and so on. **os\_cfg.h** is a required file for any  $\mu$ C/OS application. See the  $\mu$ C/OS-III (or  $\mu$ C/OS-II) documentation and books for further information.

**app.c** contains the application code for the example project. As with most C programs, code execution starts at **main()**. At a minimum, **app.c** initializes  $\mu$ C/OS and creates a startup task that initializes other Micrium modules.

**app\_cfg.h** is a configuration file for your application. This file contains **#defines** to configure the priorities and stack sizes of your application and the Micrium modules' tasks.

**app\_<module>.c** and **app\_<module>.h** These optional files contain the Micrium modules' ( $\mu$ C/TCP-IP,  $\mu$ C/FS,  $\mu$ C/USB-Host, etc) initialization code. They may or may not be present in the example projects.

## 2-4-2 INCLUDING USB HOST STACK SOURCE CODE

First, include the following files in your project from the  $\mu$ C/USB-Host source code distribution, as indicated in Figure 2-2.

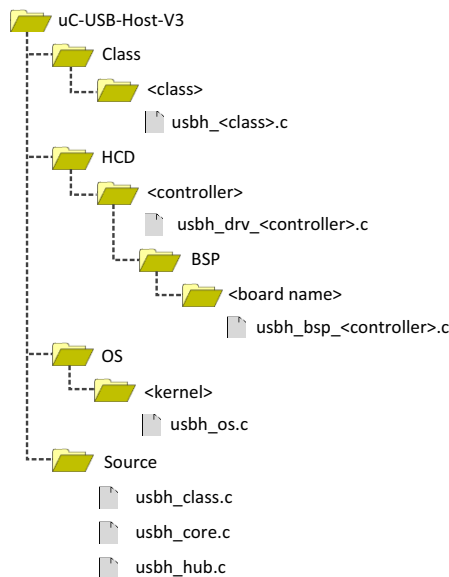


Figure 2-2  $\mu$ C/USB-Host Source Code

Second, add the following include paths to your project's C compiler settings:

```

\Micrium\Software\uC-USB-Host-V3
\Micrium\Software\uC-USB-Host-V3\HCD\<controller>\BSP\<board name>
  
```

If you are using the MSC class, you might need to add the following include path as well:

```

\Micrium\Software\uC-USB-Host-V3\Class\MSC
  
```

### 2-4-3 COPYING AND MODIFYING TEMPLATE FILES

Copy the files from the application template and configuration folders into your application as illustrated in Figure 2-3.

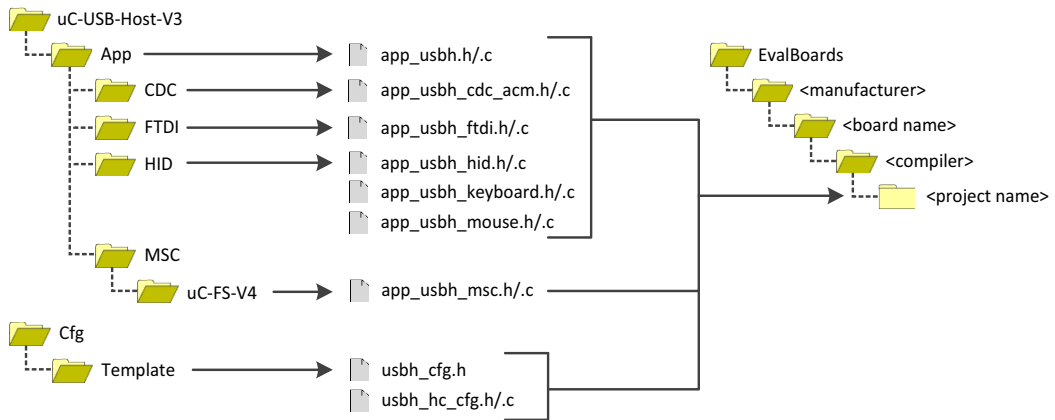


Figure 2-3 Copying Template Files.

**app\_usbh.h/.c** are the master template for USB host application-specific initialization code. This file contains the function **App\_USBH\_Init()**, which initializes the USB host stack and class-specific demos. Depending on the Kernel used, you might have to modify the USBH\_STK data type for the kernel task's stacks.

**app\_usbh\_<class>.c** contains a template to initialize and use a certain class. This file contains the class demo application. In general, the class application initializes the class, registers the class driver to the core, and performs some communication once a device is connected. Refer to the chapter(s) of the USB class(es) you purchased for more details about the class demos.

**usbh\_cfg.h** is a configuration file used to setup  $\mu$ C/USB-Host stack parameters such as the maximum number of devices, endpoints, or class-related parameters.

**usbh\_hc\_cfg.h/.c** are configuration files used to set the Host Controller Driver(s) parameters such as base address, dedicated memory base address/size and number of endpoints.

## MODIFY HOST CONTROLLER CONFIGURATION

Modify the host controller configuration file (`usbh_hc_cfg.c`) as needed for your application and USB Host Controller (HC). Refer to the driver specific readme file located in the `\Micrium\Software\uC-USB-Host-V3\HCD\<driver name>` folder for more details on how to fill this structure. Listing 2-1 describes the structure's content. Note that you should declare one structure per HC you will use.

```

USBDEVCFG USBH_HC_TemplateCfg = {           (1)
    (CPU_ADDR)0x00000000u,                 (2)
    (CPU_ADDR)0x00000000u,                 (3)
        0u,
    DEF_ENABLED,                           (4)
    1024u,                                  (5)
    2u,                                     (6)
    2u,                                     (7)
    0u                                      (8)
};

```

Listing 2-1 Host Controller Configuration Template

- L2-1(1) Give your HC configuration a meaningful name by replacing the word **“Template”**.
- L2-1(2) Specify the base address of your USB HC.
- L2-1(3) If your target has dedicated memory for the USB HC, you can specify its base address and size here. Depending on the USB HC, dedicated memory can be used to allocate driver buffers or DMA descriptors.
- L2-1(4) If the USB HC has direct access to the system memory, specify `DEF_ENABLED` here. Otherwise, `DEF_DISABLED` should be specified.
- L2-1(5) Specify the maximum length of the buffers that will be sent/received by the HC.
- L2-1(6) Specify the maximum of simultaneously opened bulk endpoints.
- L2-1(7) Specify the maximum of simultaneously opened interrupt endpoints.
- L2-1(8) Specify the maximum of simultaneously opened isochronous endpoints.

## MODIFY USB APPLICATION INITIALIZATION CODE

Listing 2-2 shows the code that you should modify based on your specific configuration done previously. You should modify the parts that are highlighted by the text in bold. The code snippet is extracted from the function `App_USBH_Init()` defined in `app_usbh.c`. The complete initialization sequence performed by `App_USBH_Init()` is presented in Listing 2-4.

```

#include <usbh_bsp_template.h>                                     (1)

CPU_BOOLEAN App_USBH_Init (void)
{
    USBH_ERR    err;
    CPU_INT08U  hc_nbr;

    err = USBH_Init(&AsyncTaskInfo,                               (2)
                   &HubTaskInfo);

    ....

    hc_nbr = USBH_HC_Add(&USBH_HC_TemplateCfg,                   (3)
                        &TemplateHCD_DrvAPI,                    (4)
                        &TemplateHCD_RH_API,                    (5)
                        &TemplateBSP_API,                       (6)
                        &err);

    err = USBH_HC_Start(hc_nbr);                                  (7)

    ....
}

```

Listing 2-2 `App_USBH_Init()` in `app_usbh.c`

L2-2(1) Include the USB driver BSP header file that is specific to your board. This file can be found in the following folder:

```

\Micrium\Software\uC-USB-Host\HCD\<controller>\BSP\<board name>

```

L2-2(2) Initialize the USB host stack's internal variables, structures and kernel port.

L2-2(3) Specify the address of the host controller configuration structure that you modified in section “Modify Host Controller Configuration” on page 29.

- L2-2(4) Specify the address of the host controller driver's API structure. The driver's API structure is defined in the driver's header file usually named `usbh_hcd_<controller>.h`.
- L2-2(5) Specify the address of the host controller driver's Root Hub (RH) API structure. The RH API structure is defined in the driver's header file usually named `usbh_hcd_<controller>.h`.
- L2-2(6) Specify the address of the host controller driver's Board Support Package (BSP) API structure. The BSP API structure is defined in the BSP header file usually named `usbh_bsp_<board name>.h`.
- L2-2(7) Starts the given host controller. If you have more than one host controller, you should first add them all and then start each of them.

## 2-4-4 MODIFYING THE APPLICATION CONFIGURATION FILE

The USB application initialization code templates assume the presence of `app_cfg.h`. The following `#defines` must be present in `app_cfg.h` in order to build the sample application.

```
#define APP_CFG_USBH_EN                DEF_ENABLED          (1)

#define USBH_OS_CFG_ASYNC_TASK_PRIO    4u                   (2)
#define USBH_OS_CFG_HUB_TASK_PRIO      3u
#define USBH_OS_CFG_ASYNC_TASK_STK_SIZE 512u
#define USBH_OS_CFG_HUB_TASK_STK_SIZE  512u

#define APP_CFG_USBH_CDC_EN             DEF_ENABLED          (3)
#define APP_CFG_USBH_FTDI_EN           DEF_ENABLED
#define APP_CFG_USBH_HID_EN            DEF_ENABLED
#define APP_CFG_USBH_MSC_EN            DEF_ENABLED

#define LIB_MEM_CFG_OPTIMIZE_ASM_EN     DEF_DISABLED         (4)
#define LIB_MEM_CFG_ARG_CHK_EXT_EN     DEF_ENABLED
#define LIB_MEM_CFG_ALLOC_EN           DEF_ENABLED
#define LIB_MEM_CFG_HEAP_SIZE          4096u

#define TRACE_LEVEL_OFF                 0u                   (5)
#define TRACE_LEVEL_INFO                1u
#define TRACE_LEVEL_DBG                 2u

#define APP_CFG_TRACE_LEVEL             TRACE_LEVEL_DBG      (6)
#define APP_CFG_TRACE                   printf               (7)

#define APP_TRACE_INFO(x)  \
((APP_CFG_TRACE_LEVEL >= TRACE_LEVEL_INFO) ? (void)(APP_CFG_TRACE x) : (void)0)
#define APP_TRACE_DBG(x)   \
((APP_CFG_TRACE_LEVEL >= TRACE_LEVEL_DBG) ? (void)(APP_CFG_TRACE x) : (void)0)
```

Listing 2-3 Application Configuration `#defines`

L2-3(1) `APP_CFG_USBH_EN` enables or disables the USB host application initialization code.

L2-3(2) These `#defines` relate to the  $\mu$ C/USB-Host kernel requirements. The  $\mu$ C/USB-Host core requires two tasks, one to manage hub requests (device connection and enumeration) and another to manage asynchronous transfers. To properly set the priority of the asynchronous and hub tasks, refer to Appendix 4, “Task Priorities” on page 55.



- L2-3(3) This `#define` enables the USB host class-specific demo. You can enable one or more USB host class-specific demos. If you enable several USB host class-specific demos, you will be able to communicate with more than one type of devices.
- L2-3(4) Configure the desired size of the heap memory. Heap memory is used by  $\mu$ C/USB-Host drivers that use internal buffers and DMA descriptors which are allocated at run-time. It is also used to allocate extra USB Request Blocks (URB) for each endpoint. Refer to the  $\mu$ C/LIB documentation for more details on the other  $\mu$ C/LIB constants.
- L2-3(5) Most Micrium examples contain application trace macros to output human-readable debugging information. Two levels of tracing are enabled: INFO and DBG. INFO traces high-level operations, and DBG traces high-level operations and return errors. Application-level tracing is different from  $\mu$ C/USB-Host tracing (refer to section 4-2 on page 54 for more details).
- L2-3(6) Define the application trace level.
- L2-3(7) Specify which function should be used to redirect the output of human-readable application tracing. You can select the standard output via `printf()`, or another output such as a text terminal using a serial interface.

Note that each class requires its own applications-specific configuration. These configurations are described in the class chapters.

## 2-5 RUNNING THE SAMPLE APPLICATION

The first step to integrate the demo application into your application code is to call `App_USBH_Init()`. This function is responsible for the following steps:

- Initializing the USB host stack.
- Calling USB host class-specific application code.
- Adding host controller(s) to the stack.
- Starting the host controller(s).

The `App_USBH_Init()` function is described in Listing 2-4.

```

#if (APP_CFG_USBH_EN == DEF_ENABLED)
CPU_BOOLEAN App_USBH_Init (void)
{
    USBH_ERR    err;
    CPU_INT08U  hc_nbr;

    err = USBH_Init(&AsyncTaskInfo,           (1)
                   &HubTaskInfo);
    if (err != USBH_ERR_NONE) {
        /* $$$$ Handle error */
        return (DEF_FAIL);
    }

#if (APP_CFG_USBH_<class>_EN == DEF_ENABLED)
    err = App_USBH_<class>_Init();           (2)
    if (err != USBH_ERR_NONE) {
        /* $$$$ Handle error */
        return (DEF_FAIL);
    }
#endif

    hc_nbr = USBH_HC_Add(&USBH_HC_TemplateCfg, (3)
                        &TemplateHCD_DrvAPI,
                        &TemplateHCD_RH_API,
                        &TemplateBSP_API,
                        &err);
    if (err != USBH_ERR_NONE) {
        /* $$$$ Handle error */
        return (DEF_FAIL);
    }

    err = USBH_HC_Start(hc_nbr);             (4)
    if (err != USBH_ERR_NONE) {
        /* $$$$ Handle error */
        return (DEF_FAIL);
    }

    return (DEF_OK);
}
#endif

```

Listing 2-4 `App_USBH_Init()` Function

- L2-4(1) `USBH_Init()` initializes the USB host stack. This must be the first USB host function called by your application's initialization code. If  $\mu$ C/USB-Host is used with  $\mu$ C/OS-II or -III, `OSInit()` must be called prior to `USBH_Init()` in order to initialize the kernel services.
- L2-4(2) Initialize the class-specific application demos by calling the function `App_USBH_<class>_Init()` where `<class>` can be `CDC`, `FTDI`, `HID` or `MSC`. Class-specific demos are enabled and disabled using the `APP_CFG_USBH_<class>_EN` #define.
- L2-4(3) `USBH_HC_Add()` creates and adds a USB host controller to the stack. If your target supports multiple host controllers, you can add multiple USB host controllers. The function `USBD_HC_Add()` returns a host controller number; this number is used as a parameter for subsequent operations on host controller(s).
- L2-4(4) After all the host controller(s) have been successfully added to the stack and the class(es) have been initialized, `USBH_HC_Start()` should be called for each host controller. This function will enable interrupts on the host controller and will start listening for device connections. If you have more than one USB host controller, pay particular attention to the order you start them. For instance, companion USB host controllers should be started *before* the main controller. See Appendix 4, "Host Controller Initialization" on page 58 for more details about companion USB host controllers.

After building and downloading the application to your target, you should be able to successfully connect a device to it through USB. Once a USB device is connected, the host stack will detect the connection of a new device and will start the enumeration process. The stack will then probe each of the classes that have been added in order to find one that can handle the device. If no driver is found for your device, you will not be able to communicate with it. Once a class driver is found, the host stack is ready to communicate with the device. Table 2-2 lists the different section(s) you should refer to for more details on each USB class demo.

<b>Class</b>	<b>Refer to...</b>
CDC ACM	See “Demo Application” on page 101.
FTDI	See “Demo Application” on page 109.
HID	See “Demo Application” on page 125.
MSC	See “Demo Application” on page 134.

Table 2-2 **USB Class Demos References**



Chapter

# 3

Architecture

$\mu$ C/USB-Host has been designed to be modular and adaptable to a variety of Central Processing Units (CPUs), real-time kernels, USB host controllers, and C compilers.

In this chapter, you will learn how the  $\mu$ C/USB-Host structure and core functionality have been implemented. It is not required that you read and understand this chapter to be able to use  $\mu$ C/USB-Host, but it will certainly help you to understand the key concepts.

### 3-1 USB HOST STACK OVERVIEW

Figure 3-1 shows an overview of the relationships between the different modules and layers of the  $\mu$ C/USB-Host stack.

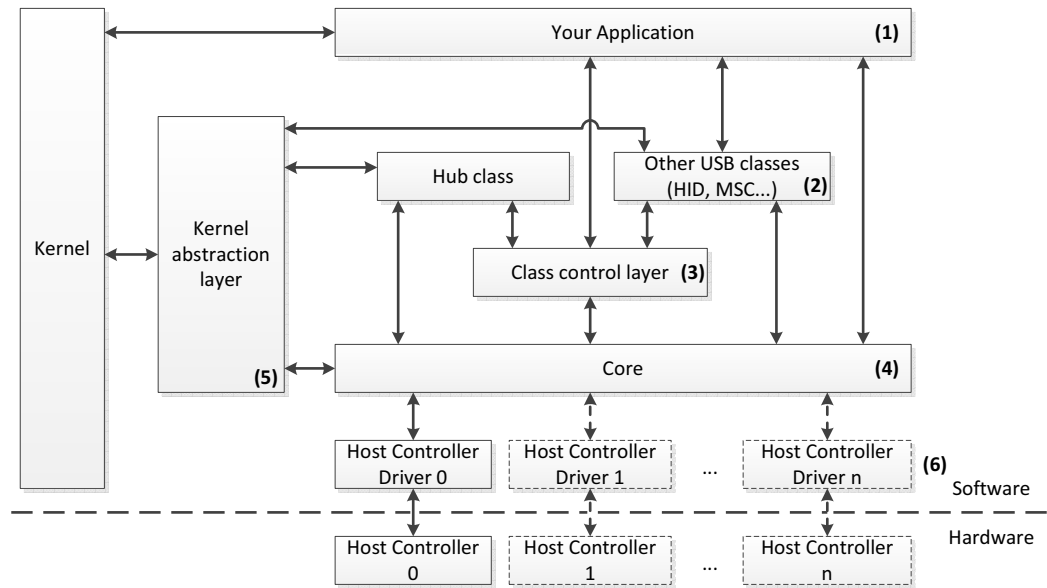


Figure 3-1 Overview of the  $\mu$ C/USB-Host Stack

- F3-1(1) For more information on the application layer, see Table 3-1.
- F3-1(2) For more information on the USB class layer, see Table 3-2.
- F3-1(3) For more information on the class control layer, see Table 3-3.
- F3-1(4) For more information on the core layer, see Table 3-4.
- F3-1(5) For more information on the kernel abstraction layer, see Table 3-5.
- F3-1(6) For more information on the Host Controller Driver layer, see Table 3-6.

## 3-1-1 USB HOST STACK MODULES

### APPLICATION

Table 3-1 summarizes the application layer characteristics.

<b>Name</b>	<b>Application layer</b>
Description	This is where your USB application is located
Interacts with	Core layer USB class layer Class control layer
Files involved	app_cfg.h                      usbh_cfg.h usbh_hc_cfg.h/.c              other files part of your application

Table 3-1 **Application Layer Characteristics**

Your application interacts with the USB classes, the class control layer, and the core layer via a set of defined APIs and callbacks. See class chapters and API reference appendices for more details about the interactions.

### USB CLASS LAYER

Table 3-2 summarizes the class layer characteristics.

<b>Name</b>	<b>USB class layer</b>
Description	This is where the class specific implementations are located
Interacts with	Application layer Core layer Kernel abstraction layer Class control layer
Files involved	usbh_hub.h/.c usbh_<class>.c/.h

Table 3-2 **USB Class Layer Characteristics**

Each class specific implementation (endpoint communication, specific protocol, data/messaging format) is located in this layer. Except for the Hub class, your application will interact with this layer to communicate with devices.



## CLASS CONTROL LAYER

Table 3-3 summarizes the class control layer characteristics.

Name	USB class control layer
Description	Manage the device function <-> class driver association
Interacts with	Application layer USB class layer Core layer
Files involved	usbh_class.h/.c

Table 3-3 **Class Control Layer Characteristics**

The class control layer is in charge of the association between a device function and a USB class driver. Your application registers the class driver(s) it needs to the class control layer during initialization. Once a device is connected and has been enumerated, the class control layer will probe each class driver until it finds a class that can handle the device or one of its functions.

## CORE LAYER

Table 3-4 summarizes the core layer characteristics.

Name	USB core layer
Description	Manage devices connection, enumeration and communication
Interacts with	Application layer Class control layer USB class layer Host Controller Driver(s) (HCD) Kernel abstraction layer
Files involved	usbh_core.h/.c

Table 3-4 **Core Layer Characteristics**

The core layer is the heart of the  $\mu$ C/USB-Host stack. This is where the device enumeration, communication and general management is handled. This is also where the general management of host controllers is handled.

---

## KERNEL ABSTRACTION LAYER

Table 3-5 summarizes the kernel abstraction layer characteristics.

Name	Kernel abstraction layer
Description	Offers an abstraction between the USB host stack and the real-time kernel.
Interacts with	USB class layer Host Controller Driver(s) (HCD)
Files involved	usbh_os.h/.c

Table 3-5 **Kernel Abstraction Layer Characteristics**

The kernel abstraction layer is a module that provides typical kernel services to the stack (task management, semaphore, mutex, etc...).  $\mu$ C/USB-Host assumes the presence of a real-time kernel, and this layer allows the USB host stack to be used with nearly any real-time kernel available. At the very least, the kernel used should provide the following services:

- Task creation at run-time
- Task delay
- Semaphore
- Mutex
- Message queue

Micrium provides a kernel abstraction layer for  $\mu$ C/OS-II and  $\mu$ C/OS-III. For more information on how to port  $\mu$ C/USB-Host to a real-time kernel, see Chapter 10, “Porting  $\mu$ C/USB-Host to your Kernel” on page 139.

## HOST CONTROLLER DRIVER LAYER

Table 3-4 summarizes the host controller driver layer characteristics.

Name	Host Controller Driver (HCD) layer
Description	Offers an abstraction between the host controller(s) hardware and the core layer.
Interacts with	Core layer
Files involved	usbh_hcd_<controller>.h/.c usbh_bsp_<controller>.h/.c

Table 3-6 Host Controller Driver Layer Characteristics

The HCD layer is an abstraction between the core layer and the host controller(s) hardware. It handles all the operations that are specific to the hardware (initialization, device and endpoints allocation/configuration, reception/transmission of USB packets, events report to the core, etc.). The USB host controller functions are encapsulated and implemented in the `usbh_hcd_<controller>.c` file. In order to have independent configuration for clock gating, interrupt controller and I/O pins control, specific to your board, another file must be implemented. This file is named `usbh_bsp_<controller>.c`, it is called the Board Specific Package (BSP). It contains everything that is closely related to the hardware on which the product will be used. Depending on the platform used, a project using  $\mu$ C/USB-Host may contain more than one HCD. For more information on how to integrate/use multiple host controllers within  $\mu$ C/USB-Host, see Chapter 4, “Configuration” on page 48. For more information on how to write a HCD, see Chapter 5, “Host Driver Guide” on page 72.

### 3-1-2 USB HOST STACK DEPENDENCIES

The USB host stack depends on other Micrium products. Following is a list of them.

#### LIBRARIES

Given that  $\mu$ C/USB-Host is designed to be used in safety critical applications, some of the “standard” library functions such as `strcpy()`, `memset()`, etc. have been rewritten to conform to the same quality standards as the rest of the USB Host stack. All these standard functions are part of a separate Micrium product called  $\mu$ C/LIB.  $\mu$ C/USB-Host depends on this product. In addition, some data objects in USB controller drivers and core layer are created at run-time which implies the use of memory allocation from the heap function `Mem_HeapAlloc()`.

## **CPU LAYER**

$\mu$ C/USB-Host can work with either an 8, 16, 32 or even 64-bit CPU, but it must have information about the CPU used. The CPU layer defines such information as the C data type corresponding to 16-bit and 32-bit variables, whether the CPU has little or big endian memory organization, and how interrupts are disabled and enabled on the CPU.

CPU-specific files are found in the `\uC-CPU` directory and are used to adapt  $\mu$ C/USB-Host to a different CPU.

## **3-2 SENDING AND RECEIVING DATA**

$\mu$ C/USB-Host is capable of sending and receiving data in two different ways: synchronously and asynchronously. In synchronous mode, the class/application blocks until the transfer operation completes or an error or time-out occurs. In asynchronous mode, the class/application does not block. An internal task of  $\mu$ C/USB-Host calls a class/application callback function once the transfer has completed. It will also inform of any error that could have occurred during the transfer. Note that for control transfers, only the synchronous mode is available.

## **3-3 TASK MODEL**

For its proper operation,  $\mu$ C/USB-Host core layer requires two tasks called *hub* and *async* task. The following sections describe these two tasks.

### **3-3-1 HUB TASK**

When a hub (root or external) reports a status change on one of its ports to the host, the host must interrogate it to discover what happened. The *hub* task is used for that process. Once a port status change is reported by a hub, a hub event is added to a queue to be further processed by the *hub* task. This task also handles the enumeration of any newly connected devices. The *hub* task is part of the hub class and is implemented as a state machine as illustrated in Figure 3-2.

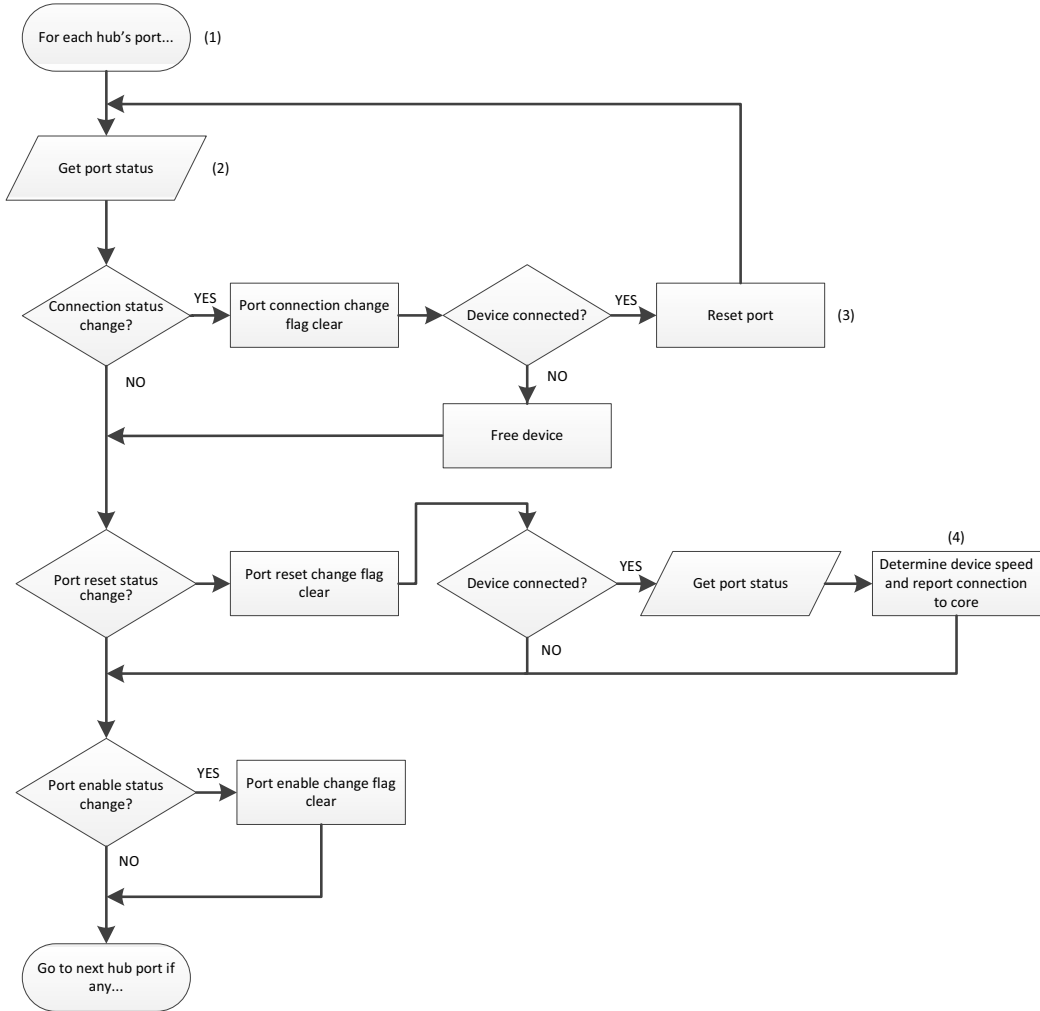


Figure 3-2 Hub Task State Machine

F3-2(1) Once the hub task is resumed, it will take the first hub event from the queue and will execute a series of operations on each of its port(s).

F3-2(2) The host requests the port status. Port status contains information like device connection status, port enable status, reset state, speed of the connected device, etc. It also indicates which field changed since the last port status report. For more information on port status, see the Universal Serial Bus specification, revision 2.0, section 11.24.2.7.

F3-2(3) When a device is connected, the first thing to do is to reset it. Once the device has been properly reset, the hub task will request the port status once again. The new port status should indicate that the port reset status changed.

F3-2(4) At this point, the device has been reset and the necessary resources to handle it have been allocated by the stack. It is now time to inform the core of the presence of a new device so it can enumerate it. For more information on how the core enumerates a newly connected device, see section 3-4 “Enumeration Process” on page 46.

### 3-3-2 ASYNCHRONOUS TASK

This task manages all the asynchronous transfers that have been initiated by the class or the application. Once the transfer has completed, the core will be notified of the completion by the Host Controller Driver (HCD). If the transfer was initiated with an asynchronous API, this task is resumed and will call the class/application callback function. Note that this task is never used with control transfers as no asynchronous APIs are available for this type of transfer.

## 3-4 ENUMERATION PROCESS

Once a device has been connected and recognized/allocated by the *hub* task (as described in section 3-3-1 “Hub Task” on page 44), the device will be enumerated by the core. The enumeration is a process by which the host learns about the device’s capabilities and functions. Figure 3-3 summarizes the enumeration process as executed by the  $\mu$ C/USB-Host stack.

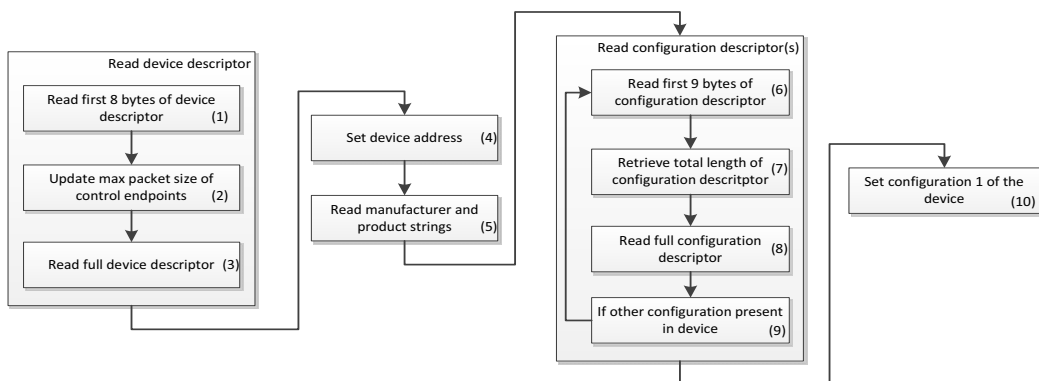


Figure 3-3 Enumeration Process as Executed by  $\mu$ C/USB-Host

- F3-3(1) First step consists in reading the first 8 bytes of the device descriptor. At this point, the maximum packet size of the device control endpoints is unknown to the core, hence it is impossible to read more than 8 bytes.
- F3-3(2) The core retrieves and updates the maximum packet size of the control endpoints from the information contained in the first 8 bytes of the device descriptor.
- F3-3(3) Now that the core knows what is the maximum packet size of the control endpoints, it can request the full device descriptor.
- F3-3(4) Since the beginning, the device always responded to the default address 0. At this point, the core gives the device a unique address via a *SetAddress* request.
- F3-3(5) If the manufacturer and product string index contained in the device descriptor are non-zero, the core requests their content. This is only useful for tracing and debugging purposes.
- F3-3(6) At this moment, the core must read the configuration descriptor(s). Since the total length of the configuration descriptor is unknown, it is necessary to read the 9 first bytes of the configuration descriptor to retrieve its total length.
- F3-3(7) With the partial configuration descriptor, the core is now aware of the total length and knows how many bytes should be requested from the device.
- F3-3(8) The core reads the entire configuration descriptor.
- F3-3(9) If another configuration is available on the device, the core will read its descriptor as well. It will then have to go back to step (6).
- F3-3(10) Once all the configuration descriptors have been read, the core will, by default, set the first configuration. You are now ready to communicate with the device via your application.

Before running it,  $\mu$ C/USB-Host you must configure it properly. There are three groups of configuration parameters:

- Static stack configuration
- Application specific configuration
- Host Controller Driver (HCD) configuration

This chapter explains how to setup all these groups of configuration. The last section of this chapter also provides examples of configuration following examples of typical use.

## 4-1 STATIC STACK CONFIGURATION

$\mu$ C/USB-Host is configurable at compile time via approximately 30 **#defines** in the application's copy of **usbh\_cfg.h**.  $\mu$ C/USB-Host uses **#defines** when possible, because they allow code and data sizes to be scaled at compile time based on enabled features and the configured number of USB objects. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of  $\mu$ C/USB-Host to be adjusted based on application requirements.

It is recommended that the configuration process begins with the default configuration values which in the next sections will be shown in **bold**.

The sections in this chapter are organized following the order in  $\mu$ C/USB-Host's template configuration file, **usbh\_cfg.h**.



---

## 4-1-1 USB HOST CONFIGURATION

### USBH\_CFG\_MAX\_NBR\_DEVS

**USBH\_CFG\_MAX\_NBR\_DEVS** configures the maximum number of devices supported by the  $\mu$ C/USB-Host stack. USB uses a tiered star topology. Up to five external hubs can connect in series with a limit of 127 peripherals and hubs including the root hub. There is one root hub per USB host controller. **USBH\_CFG\_MAX\_NBR\_DEVS** value should always be less than 127 physical devices. If the host accepts only one single device with no use of external hubs, the value may be set to 1. Default value is **4**.

### USBH\_CFG\_MAX\_NBR\_CFGS

**USBH\_CFG\_MAX\_NBR\_CFGS** sets the maximum number of USB configurations per USB device. Most of the commercial USB devices support only one configuration. Refer to the *Universal Serial Bus specification, Revision 2.0, section 9.2.3* for more details about USB configuration. Default value is **1**.

### USBH\_CFG\_MAX\_NBR\_IFS

**USBH\_CFG\_MAX\_NBR\_IFS** configures the maximum number of interfaces available per configuration. This value greatly depends on the USB class(es) used. Most of the supported classes require at least one interface, while CDC ACM requires two. Refer to the *Universal Serial Bus specification, Revision 2.0, section 9.2.3* for more details about USB interfaces. Default value is **2**. If a composite/compound device is to be attached to the  $\mu$ C/USB-Host stack, you may need to increase this value to 3 or 4.

### USBH\_CFG\_MAX\_NBR\_EPS

**USBH\_CFG\_MAX\_NBR\_EPS** configures the maximum number of allowed endpoints per interface alternate setting. This value greatly depends on the USB class(es) used. For information on how many endpoints are needed for each class, refer to the class specific chapter. In general, an interface needs a pair of non-control endpoints for bidirectional communication. Default value is **2**.

---

## **USBH\_CFG\_MAX\_NBR\_CLASS\_DRVS**

**USBH\_CFG\_MAX\_NBR\_CLASS\_DRVS** configures the maximum number of class drivers (HID, MSC, etc.) supported by the  $\mu$ C/USB-Host stack. The value should be greater or equal to 2 because the  $\mu$ C/USB-Host stack always integrates the Hub class counting for one class driver. Thus the minimum value would be one Hub class + one of the supported classes. The default value is **4**.

## **USBH\_CFG\_MAX\_CFG\_DATA\_LEN**

**USBH\_CFG\_MAX\_CFG\_DATA\_LEN** configures the maximum length of the buffer dedicated to receive the entire configuration descriptor content sent by the device. The full configuration descriptor contains the configuration descriptor itself, all Interfaces and their associated endpoint descriptors and any class-specific descriptors. Default value is **256**. This value is convenient for standard classes such as CDC ACM, HID and MSC.

## **USBH\_CFG\_MAX\_HUBS**

**USBH\_CFG\_MAX\_HUBS** configures the maximum number of hubs supported by the  $\mu$ C/USB-Host stack. The hubs encompass external hubs and root hub(s). In general, there is one root hub per host controller (HC). Default value is **2** (1 root hub + 1 external hub).

## **USBH\_CFG\_MAX\_HUB\_PORTS**

**USBH\_CFG\_MAX\_HUB\_PORTS** specifies the maximum number of ports that can be active and managed at the same time per external hub connected to the host. If **USBH\_CFG\_MAX\_HUB\_PORTS** is set to a value less than the actual number of ports available on a hub, then the rest of the ports on this hub will be rendered unusable. For instance, if the value is 4 and the hub has 7 ports, 3 ports are inactive. If you connect a device to one of these 3 inactive ports, the host stack will not detect the connection. Currently on the market, the number of ports for external hubs can range from 2 to 7 with the most common being 4 and 7. Default value is **7**.

Figure 4-1 shows two 7-port hubs. One of them has a peculiarity explained below.

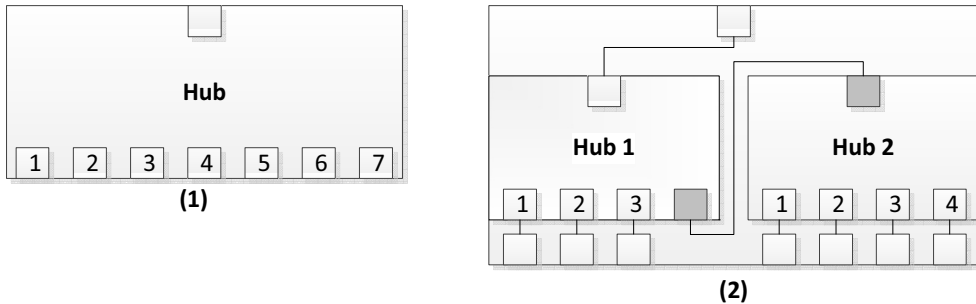


Figure 4-1 7-Port Hubs

- F4-1(1) This 7-port hub is a normal external hub. The host will identify this hub as one unique device.
- F4-1(2) For the user, this 7-port hub looks externally the same as the one on the left, that is one upstream port connecting to the host and 7 downstream ports connecting to devices. In fact, this hub is composed of two hubs in series. Internally, the fourth port of Hub 1 is connected to the upstream port of Hub 2. You may encounter commercial 7-port external hubs that are designed internally as shown on the right hub. When this hub is connected to  $\mu$ C/USB-Host, the stack will see two devices. It will enumerate twice the hub: one enumeration for Hub 1 and one enumeration for Hub 2. Thus, you may need to increase `USBH_CFG_MAX_NBR_DEVS` to avoid reaching the maximum number of supported devices.

### **USBH\_CFG\_MAX\_STR\_LEN**

`USBH_CFG_MAX_STR_LEN` specifies the maximum buffer length aimed to receive a string retrieved from the device as part of a *GetDescriptor(String)* request. If logging is not enabled (constant `USBH_CFG_PRINT_LOG` set to `DEF_DISABLED`), then `USBH_CFG_MAX_STR_LEN` can be set to 0. Default value is **256**.

---

### **USBH\_CFG\_STD\_REQ\_TIMEOUT**

**USBH\_CFG\_STD\_REQ\_TIMEOUT** specifies a timeout expressed in milliseconds used during control transfers. This timeout defines the time allowed for the device to complete the standard request initiated by the host. Default value is **5000**. Refer to the *Universal Serial Bus specification, Revision 2.0, section 9.2.6.4* for more details about timeouts during standard device requests.

### **USBH\_CFG\_STD\_REQ\_RETRY**

**USBH\_CFG\_STD\_REQ\_RETRY** specifies the maximum number of attempts to get a certain descriptor. Default value is **3**.

### **USBH\_CFG\_MAX\_NBR\_HC**

**USBH\_CFG\_MAX\_NBR\_HC** specifies the maximum number of Host Controllers (HC) supported by the  $\mu$ C/USB-Host stack. Default value is **1**.

### **USBH\_CFG\_MAX\_ISOC\_DESC**

**USBH\_CFG\_MAX\_ISOC\_DESC** configures the maximum number of isochronous descriptors available for all isochronous endpoints. Default value is **1**.

### **USBH\_CFG\_MAX\_EXTRA\_URB\_PER\_DEV**

**USBH\_CFG\_MAX\_EXTRA\_URB\_PER\_DEV** configures the maximum number of additional USB Request Blocks (URB) available to perform streaming communication using asynchronous mode. Default value is **1**.

## **4-1-2 USB CLASSES CONFIGURATION**

Refer to USB class chapters for more details about configuration constants related to each USB class.

---

### 4-1-3 DEBUG CONFIGURATION

Configurations in this section only need to be set during development/debug stages. Debug configuration allows you to output useful debug messages on a serial terminal or a debugger console. A set of debug macros are used in the different layers of  $\mu$ C/USB-Host. Different levels of debug messages can be chosen.

#### **USBH\_CFG\_PRINT\_LOG**

**USBH\_CFG\_PRINT\_LOG** enables or disables the functions to print messages to debug or trace the stack's flow of execution. When set to **DEF\_ENABLED**, this configuration constant defines the following macro: **USBH\_PRINT\_LOG()**. This macro is used in different layers of  $\mu$ C/USB-Host and gives you debug information specific to a certain class, the core and the driver. Default value is **DEF\_DISABLED**.

#### **USBH\_CFG\_PRINT\_ERR**

**USBH\_CFG\_PRINT\_ERR** enables or disables the debug messages that indicate a specific error code. When set to **DEF\_ENABLED**, this configuration constant defines the following macro: **USBH\_PRINT\_ERR()**. This macro is used in different layers of  $\mu$ C/USB-Host and gives you error codes specific to a certain class or the core. Default value is **DEF\_DISABLED**.

#### **USBH\_PRINTF**

**USBH\_PRINTF** macro allows you to output your debug messages on a debugger console or serial terminal by mapping **USBH\_PRINTF** to a certain **printf()** function or equivalent. **USBH\_PRINTF** is used by the macros **USBH\_PRINT\_LOG()** and **USBH\_PRINT\_ERR()** as shown in Code Listing 4-1.

```
#include <stdio.h>
#define USBH_PRINTF          printf                (1)

#if (USBH_CFG_PRINT_LOG == DEF_ENABLED)
#define USBH_PRINT_LOG(...)  USBH_PRINTF(__VA_ARGS__) (2)
#endif

#if (USBH_CFG_PRINT_ERR == DEF_ENABLED)          (3)
#define USBH_PRINT_ERR(err)  USBH_PRINTF("ERR:%s:%d:err=%d\n", __FUNCTION__, __LINE__, err);
#else
#define USBH_PRINT_ERR(err)
#endif
```

Listing 4-1 Debug Macros

- L4-1(1) In this example, `USBH_PRINTF` is mapped to a standard `printf()` function defined in the standard I/O library, `stdio.h`, provided by the toolchain. Debug messages will be outputted on the debugger's console window.
- L4-1(2) If `USBH_CFG_PRINT_LOG` is set to `DEF_ENABLED`, informative messages will be outputted to the console using the classic formatting allowed by the `printf()` function.
- L4-1(3) If `USBH_CFG_PRINT_ERR` is set to `DEF_ENABLED`, any error will be outputted to the console using a formatting such that the name of the function in which the error has been flagged, the line number at which the error was detected and the error code will be displayed.

## 4-2 APPLICATION SPECIFIC CONFIGURATION

This section defines the configuration constants related to  $\mu$ C/USB-Host but that are application-specific. All these configuration constants relate to the kernel. For many kernels, the  $\mu$ C/USB-Host task priorities and stack sizes will need to be explicitly configured for the particular kernel (consult the specific kernel's documentation for more information).

These configuration constants should be defined in an application's `app_cfg.h` file.

---

## 4-2-1 TASK PRIORITIES

As mentioned in section 3-3 “Task Model” on page 44,  $\mu$ C/USB-Host needs one *hub* task and one *async* task for its proper operation. The priority of  $\mu$ C/USB-Host’s *hub* and *async* task greatly depends on the USB requirements of your application. For some applications, it might be better to set the *hub* task at a high priority, especially if your application requires a lot of tasks and is CPU intensive. The *hub* task is responsible for the device’s connection/disconnection detection and enumeration. In that case, if the *hub* task has a low priority, the device connection will still be detected and it will be enumerated but the notification of a device enumeration completed to your application may be delayed. The  $\mu$ C/USB-Host stack allows the classes or your application tasks to exchange data in synchronous or asynchronous mode. The  $\mu$ C/USB-Host *async* task is in charge of the asynchronous transfers. In the case where the classes use or your application deals with asynchronous transfers, it is a good idea to set the *async* task with a higher priority than the *hub* task. You may want to be notified about an asynchronous transfer completion before a new device connection. The *async* task could have a lower priority than your application tasks if you know you will have quite a lot of code in your asynchronous callback functions. For more details about synchronous and asynchronous communication, refer to section 3-2 “Sending and Receiving Data” on page 44.

For the  $\mu$ C/OS-II and  $\mu$ C/OS-III RTOS ports, the following constants must be configured within `app_cfg.h`:

- `USBH_OS_CFG_HUB_TASK_PRIO`
- `USBH_OS_CFG_ASYNC_TASK_PRIO`

## 4-2-2 TASK STACK SIZES

For the  $\mu$ C/OS-II and  $\mu$ C/OS-III RTOS ports, the following constants must be configured within `app_cfg.h` to set the internal task stack sizes:

- `USBH_OS_CFG_HUB_TASK_STK_SIZE`                    **512**
- `USBH_OS_CFG_ASYNC_TASK_STK_SIZE`                **512**

The arbitrary stack size of **512** is a good starting point for most applications.

---

The only guaranteed method of determining the required task stack sizes is to calculate the maximum stack usage for each task. Obviously, the maximum stack usage for a task is the total stack usage along the task's most-stack-greedy function path. Note that the most-stack-greedy function path is not necessarily the longest or deepest function path.

The easiest and best method for calculating the maximum stack usage for any task/function should be performed statically by the compiler or by a static analysis tool since these can calculate function/task maximum stack usage based on the compiler's actual code generation and optimization settings. So for optimal task stack configuration, we recommend to invest in a task stack calculator tool compatible with your build toolchain.

## 4-3 HOST CONTROLLER DRIVER CONFIGURATION

During the initialization of the  $\mu$ C/USB-Host stack, you will have to initialize all the host controllers of your platform. In order to do that, for each host controller, you need to:

- 1 Declare a configuration structure of type `USBH_HC_CFG` containing all the characteristics of the host controller.
- 2 Add the host controller to the stack using the function `USBH_HC_Add()`.
- 3 Start the host controller operations by calling the function `USBH_HC_Start()`.

### 4-3-1 HOST CONTROLLER CONFIGURATION STRUCTURE

The host controller configuration structure is declared in `usbh_hc_cfg.h` and defined in the file `usbh_hc_cfg.c` (refer to section 2-4-3 "Copying and Modifying Template Files" on page 28 for an example of initializing this structure). These files are distributed as templates, and you must modify them to have the proper configuration for your USB Host Controller (HC). A reference to the structure `USBH_HC_CFG` needs to be passed to the `USBH_HC_Add()` function, which allocates a HC. Each host controller driver comes with a specific readme file located in `\Micrium\Software\uC-USB-Host-V3\HCD\<driver name>` giving more information about the host controller specificities and limitations. You should refer to it to help you configure the structure `USBH_HC_CFG`.

The fields of the following structure presented in Listing 4-2 are the parameters needed to configure the USB Host Controller Driver (HCD):



```

USBH_HC_CFG USBH_HC_OHCI_Cfg_AT91SAM9M10 = {
    (CPU_ADDR)0x00700000,           (1)
    (CPU_ADDR)0,                   (2)
    0,                             (3)
    DEF_ENABLED,                   (4)
    1024u,                          (5)
    10u,                            (6)
    10u,                            (7)
    0u                               (8)
};

```

Listing 4-2 USB Host Controller Driver Configuration Structure

- L4-2(1) Base address of the USB HC hardware registers
- L4-2(2) Base address of the USB HC dedicated memory.
- L4-2(3) Size of the USB HC dedicated memory.
- L4-2(4) Flag indicating if the HC can access the system memory from which application/class/core data buffers have been allocated. If the flag is set to **DEF\_DISABLED**, buffers will be copied from system memory to the dedicated memory accessed by the HC. This flag applies to HCs and their associated drivers that support DMA.
- L4-2(5) Maximum buffer length used to receive and send data. If the field **DataBufFromSysMemEn** is set to **DEF\_ENABLED**, the host controller cannot access the system memory. Thus the HCD will allocate an associated data buffer from the HC's dedicated memory and pass the buffer information to the HC for processing. If the field **DataBufFromSysMemEn** is set to **DEF\_DISABLED**, the HCD will directly pass the buffer allocated from the system memory to the HC.
- L4-2(6) Maximum number of open bulk endpoints.
- L4-2(7) Maximum number of open interrupt endpoints.
- L4-2(8) Maximum number of open isochronous endpoints.

### 4-3-2 HOST CONTROLLER INITIALIZATION

Once the Host Controller (HC) configuration structure is ready, you will be able to add your HC to the  $\mu$ C/USB-Host stack and to start its operations. Figure 4-2 and Figure 4-3 introduce some typical HC architectures within a microcontroller.

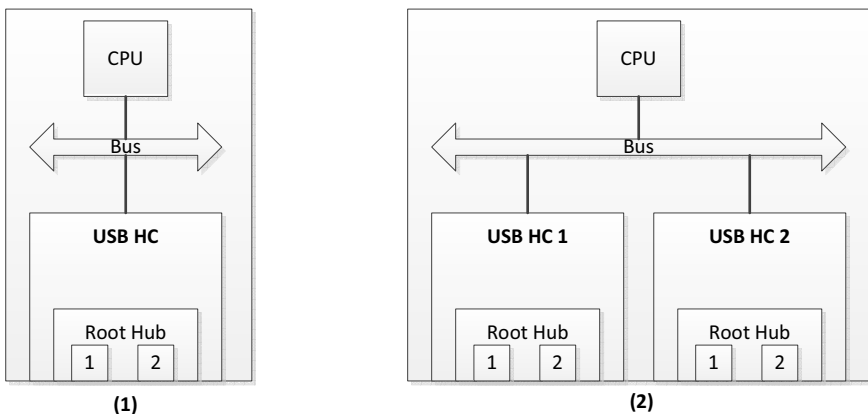


Figure 4-2 Typical Host Controller Architectures

- F4-2(1) The single HC is the most widespread architecture among embedded USB hosts. The single HC can be a vendor-specific, an Enhanced Host Controller Interface (EHCI) or an Open Host Controller Interface (OHCI) controller.
- F4-2(2) Some embedded systems contain several HCs. Here, this multi-host architecture has two HCs. Each one manages its own root hub composed of two ports.

Figure 4-3 gives another example of a multi-host architecture. Here, a main HC shares a root hub with one or more other HCs. The other HCs are called *Companion* controllers. You may find this type of architecture with EHCI and OHCI controllers where the main host controller would be an EHCI controller and companion controllers would be OHCI controllers. In general, an embedded systems' microcontroller will have one EHCI associated with one OHCI. An EHCI controller can only be in charge of high-speed devices. Low- and full-speed devices are handled by the OHCI controller(s). Refer respectively to *Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0, March 12 2002* and *OpenHCI Specification for USB, 09/14/99, Release 1.0a* for more details about EHCI and OHCI controllers respectively.

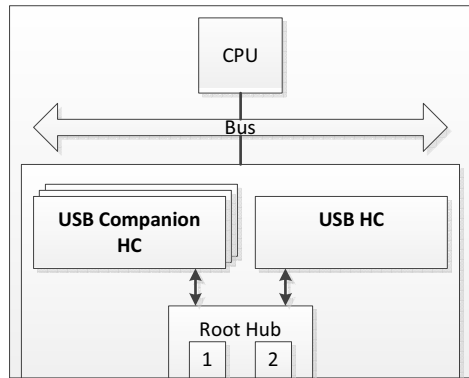


Figure 4-3 Multi-Host Architecture

The template file `app_usbh.c` shows an example of HC initialization in the function `App_USBH_Init()`. Listing 4-3 presents the body of this function. The example in this function corresponds to a single HC architecture as shown in Figure 4-2 (left drawing). The example can be easily extended to describe the multi-host architecture shown in Figure 4-2 (right drawing).

```

CPU_BOOLEAN App_USBH_Init (void)
{
    USBH_ERR    err;
    CPU_INT08U  hc_nbr;

    ...

    hc_nbr = USBH_HC_Add(&USBH_HC_TemplateCfg,           (1)
                        &TemplateHCD_DrvAPI,           (2)
                        &TemplateHCD_RH_API,           (3)
                        &TemplateBSP_API,              (4)
                        &err);                          (5)

    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    err = USBH_HC_Start(hc_nbr);                        (6)
    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    return (DEF_OK);
}

```

Listing 4-3 Single Host Controller Initialization

- 
- L4-3(1) Add the unique HC to the stack by calling `USBH_HC_Add()`. This function will allocate and initialize certain internal structures associated to the HC and used by the core layer to manage the HC. It will also initialize the controller itself and get its speed. Several parameters are passed to `USBH_HC_Add()`. `USBH_HC_Add()` will return a HC number. In case of a multi-host architecture with two HCs, the second controller is added to the stack by calling one more time `USBH_HC_Add()`. `USBH_HC_Add()` will return a HC number unique to this second HC.
- L4-3(2) `USBH_HC_TemplateCfg` is the HC configuration structure of type `USBH_HC_CFG` previously initialized in `usbh_hc_cfg.c`.
- L4-3(3) `TemplateHCD_DrvAPI` is the HC driver's API structure of type `USBH_HC_DRV_API` defined in the HC driver header file, `usbh_hcd_<controller>.h`. It defines the functions of the HC driver enabling the USB host stack core to talk to the controller. Refer to section "USB Controller API" on page 73 for more details about the `USBH_HC_DRV_API` structures and the associated API.
- L4-3(4) `TemplateHCD_RH_API` is the HC driver's Root Hub (RH) API structure of type `USBH_HC_RH_API` defined in the HC driver header file, `usbh_hcd_<controller>.h`. It contains some functions permitting the root hub management. Refer to section "Root Hub API" on page 75 for more details about the `USBH_HC_RH_API` structures and the associated API.
- L4-3(5) `TemplateBSP_API` is the HC driver's Board Support Package (BSP) API structure of type `USBH_HC_BSP_API` defined in the BSP header file, `usbh_bsp_<controller>.h`. It contains functions enabling the driver to configure the BSP part (that is USB clock, USB interrupt handler setup, etc.). Refer to section 5-6 "CPU and Board Support" on page 81 for more details about the `USBH_HC_BSP_API` structures and the associated API.
- L4-3(6) Start the USB operations of the HC. `USBH_HC_Start()` is called by passing the HC number previously obtained. This function will basically enable any interrupts that let the HC know about any root hub port events (for example, device connection/disconnection). In case of a multi-host architecture with two HCs, the second controller is started by calling once again `USBH_HC_Start()` with the correct HC number.

Listing 4-4 illustrates a multi-host initialization corresponding to Figure 4-3. In this example, a multi-host architecture composed of one EHCI and one OHCI is considered.

```

CPU_BOOLEAN App_USBH_Init (void)
{
    USBH_ERR    err;
    CPU_INT08U  ohci_hc_nbr;
    CPU_INT08U  ehci_hc_nbr;

    ...

    ehci_hc_nbr = USBH_HC_Add(&USBH_HC_EHCI_Cfg_Template,      (1)
                             &EHCI_DrvAPI,
                             &EHCI_RH_API,
                             &Template_EHCI_BSP_API,
                             &err);
    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    ohci_hc_nbr = USBH_HC_Add(&USBH_HC_OHCI_Cfg_Template,    (2)
                             &OHCI_DrvAPI,
                             &OHCI_RH_API,
                             &Template_OHCI_BSP_API,
                             &err);
    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    err = USBH_HC_Start(ohci_hc_nbr);                       (3)
    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    err = USBH_HC_Start(ehci_hc_nbr);                       (4)
    if (err != USBH_ERR_NONE) {
        return (DEF_FAIL);
    }

    return (DEF_OK);
}

```

Listing 4-4 **Multi-Host Controllers Initialization**

L4-4(1) Add the EHCI controller to the  $\mu$ C/USB-Host stack by specifying its characteristics configuration, its associated driver API, root hub API and BSP API.

- 
- L4-4(2) Add the OHCI controller to the  $\mu$ C/USB-Host stack by specifying its characteristics configuration, its associated driver API, root hub API and BSP API.
- L4-4(3) Start the OHCI controller operations.
- L4-4(4) Start the EHCI controller operations.

In the case of multi-host controllers initialization, it is recommended to *add first* all your HCs to the  $\mu$ C/USB-Host stack and *then to start* their operations. It is especially important for an EHCI controller working with OHCI controllers and sharing the same root hub. By default, the EHCI controller has the ownership of the root hub's ports. If the device connected to a certain port cannot be managed by the EHCI controller, the port ownership goes to one of the OHCI controllers. As the EHCI controller relies on OHCI controllers, OHCI controllers should be started before the EHCI controller.

For vendor-specific HCs, you might change the add/start host controller order for: adding the first controller to the stack and starting it with `USBH_HC_Start()`, then adding a second HC and starting it, and so on. In general, a vendor-specific controller will not share the same root hub with another vendor-specific controller. But adding all your HCs followed by starting each HC is a good practice.

## 4-4 CONFIGURATION EXAMPLES

This section provides examples of configuration for  $\mu$ C/USB-Host stack based on some bus topology configurations. This section will only give examples of static stack configuration, as the application-specific configuration greatly depends on your application. Also, the host controller driver configuration depends on the hardware you use.

The following examples are presented:

- A single host controller and a unique USB device connected to it.
- A single host controller and multiple USB devices connected to it.
- Multi-host controllers and multiple USB devices connected to them.

### 4-4-1 SINGLE HOST CONTROLLER AND UNIQUE DEVICE

This example is the most simple bus topology you can find in which one USB device is connected to the only port of the host controller. In our example, the USB device is a Mass Storage Class (MSC) device. Figure 4-4 introduces the first bus topology example. The platform has only one host controller whose root hub is composed of 1 port. A MSC device composed of a pair of bulk endpoints to communicate is connected to this only port.

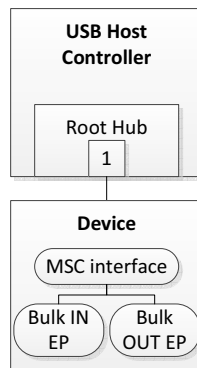


Figure 4-4 **Single Host Controller and Unique Device Configuration**

Table 4-1 shows the values that should be set for the different configuration constants described earlier in order to support this bus topology.

Configuration	Value	Explanation
USBH_CFG_MAX_NBR_DEVS	1	Only one USB device accepted at a time.
USBH_CFG_MAX_NBR_CFGS	1	Only one configuration is needed for a MSC device.
USBH_CFG_MAX_NBR_IFS	1	Only one interface is required to describe a MSC function.
USBH_CFG_MAX_NBR_EPS	2	MSC interface requires two bulk endpoints to communicate.
USBH_CFG_MAX_NBR_CLASS_DRV	2	Hub class always present in the USB Host stack + MSC class
USBH_CFG_MAX_CFG_DATA_LEN	50	Size of a Configuration Descriptor sent by a MSC device must be less than 50 bytes. In general, a full MSC Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + 2 * size(Endpoint Descriptor) = 9 + 9 + 2 * 7 = 32 bytes. You should consider adding a safety margin in case the MSC Configuration Descriptor exceeds 32 bytes (e.g. 50 bytes).

Configuration	Value	Explanation
USBH_CFG_MAX_HUBS	1	Only one hub (root hub) supported.
USBH_CFG_MAX_HUB_PORTS	1	The root hub has only one port.
USBH_CFG_MAX_STR_LEN	256	This value ensures that the USB Host stack can get any length of strings characters sent by the MSC device. This value could be lower. In general, any string describing the device is less than 50 characters.
USBH_CFG_STD_REQ_TIMEOUT	5000	The device has 5 seconds to complete a standard request sent by the host.
USBH_CFG_STD_REQ_RETRY	3	If a standard request fails, the host attempts up to 3 times to get a successful request.
USBH_CFG_MAX_NBR_HC	1	One host controller on this platform.
USBH_CFG_MAX_ISOC_DESC	1	Even if the MSC device does not use isochronous endpoints but only control and bulk endpoints, this value must be set to 1.
USBH_CFG_MAX_EXTRA_URB_PER_DEV	1	This value should be set at least to 1.

Table 4-1 Constant Values for Single Host Controller and Unique Device Configuration Example

## 4-4-2 SINGLE HOST CONTROLLER AND MULTIPLE DEVICES

This example adds a bit of complexity to the basic example previously shown in Figure 4-4. This time, the USB Host stack has to support up to 2 external hubs in series and a certain number of USB devices. The platform still contains only one host controller with one port. The bus topology presented in Figure 4-5 is composed of:

- 2 external hubs with 4 ports each. Each external hub is composed of one interrupt endpoint used for port events notification.
- 3 Mass Storage Class (MSC) devices. Each MSC device is composed of a pair of bulk endpoints to communicate.
- 2 Human Interface Devices (HID). Each HID is composed of one interrupt endpoint to communicate.

An embedded host with limited capabilities can support such bus topology with a certain depth of hubs in series and a few USB devices at the same time.



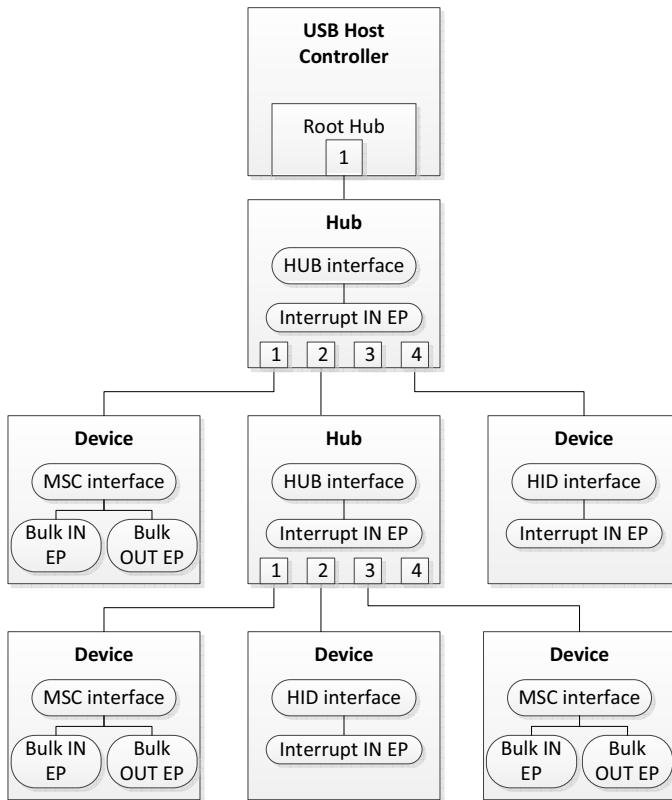


Figure 4-5 Single Host Controller and Multiple Devices Configuration

Table 4-2 shows the values that should be set for the different configuration constants described earlier in order to support this bus topology.

Configuration	Value	Explanation
USBH_CFG_MAX_NBR_DEVS	7	2 external hubs + 5 other USB devices (in our example, 3 MSC and 2 HID). An external hub is also considered as a device. $\mu$ C/USB-Host enumerates an external hub as it would do with any other USB devices. Notice that if the maximum number of supported devices has been reached, the stack will simply disregard any other USB devices trying to connect. In that case, a message could be displayed to the user saying that the maximum number of devices has been reached.
USBH_CFG_MAX_NBR_CFGS	1	Only one configuration is usually needed for an external hub, a MSC device or a HID device.

Configuration	Value	Explanation
USBH_CFG_MAX_NBR_IFS	1	Only one interface is required to describe a Hub, a MSC or a HID function.
USBH_CFG_MAX_NBR_EPS	2	The maximum number of endpoints needed among all the interfaces in this bus topology is taken. MSC interface requires two bulk endpoints to communicate. Hub and HID interfaces use only one endpoint for communication.
USBH_CFG_MAX_NBR_CLASS_DRVS	3	Hub class (always present in the USB Host stack) + MSC class + HID class.
USBH_CFG_MAX_CFG_DATA_LEN	100	Size of a Configuration Descriptor sent by an external hub, a MSC or HID device must be less than 100 bytes. In general: A full HUB Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + size(Hub Descriptor) + 1* size(Endpoint Descriptor) = 9 + 9 + 9 + 1 * 7 = 34 bytes. A full MSC Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + 2 * size(Endpoint Descriptor) = 9 + 9 + 2 * 7 = 32 bytes. <b>A full HID Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + size(HID Descriptor) + 1* size(Endpoint Descriptor) = 9 + 9 + 12 + 1 * 7 = 37 bytes.</b> Among HUB, MSC and HID, HID has the longest configuration descriptor. The value of USBH_CFG_MAX_CFG_DATA_LEN should be at least equal to the longest configuration descriptor size. A safety margin should be added. It is particularly true here because the HID descriptor has a size that varies according to the number of reports contained in the HID device. The HID descriptor size is given by this formula (9 + (N * 3)) with N = number of reports.
USBH_CFG_MAX_HUBS	3	1 root hub + 2 external hubs in series.
USBH_CFG_MAX_HUB_PORTS	4	It should always be set to the maximum of ports on the hub.
USBH_CFG_MAX_STR_LEN	256	This value ensures that the USB Host stack can get any length of strings of characters sent by any of USB devices. This value could be lower. In general, any string describing the device is less than 50 characters.
USBH_CFG_STD_REQ_TIMEOUT	5000	The device has 5 seconds to complete a standard request sent by the host.
USBH_CFG_STD_REQ_RETRY	3	If a standard request fails, the host attempts up to 3 times to get a successful request.
USBH_CFG_MAX_NBR_HC	1	One host controller on this platform.

Configuration	Value	Explanation
USBH_CFG_MAX_ISOC_DESC	1	Even if external hubs, MSC and HID devices do not use isochronous endpoints, this value must be set to 1.
USBH_CFG_MAX_EXTRA_URB_PER_DEV	1	This value should be set at least to 1.

Table 4-2 Constant Values for Single Host Controller and Multiple Devices Configuration Example

### 4-4-3 MULTI-HOST CONTROLLERS AND MULTIPLE DEVICES

This example describes a more complex bus topology. Here, the USB Host stack has to support up to 5 external hubs in series and a certain number of USB devices. The platform contains 2 host controllers. Each root hub associated to one host controller provides 2 ports. The bus topology presented in Figure 4-6 is composed of:

- 6 external hubs with 4 ports each. Among the 6 external hubs, up to 5 hubs are in series which is the maximum allowed by the USB 2.0 specification. Each external hub is composed of one interrupt endpoint used for port events notification.
- 6 Mass Storage Class (MSC) devices. Each MSC device is composed of a pair of bulk endpoints to communicate.
- 3 Human Interface Devices (HID). Each HID is composed of one interrupt endpoint to communicate.
- 2 composite devices composed of one HID function and one MSC function.

This bus topology requires an embedded host with good capabilities to support the maximum depth of hubs in series and several USB devices connected at the same time to root hubs and external hubs.

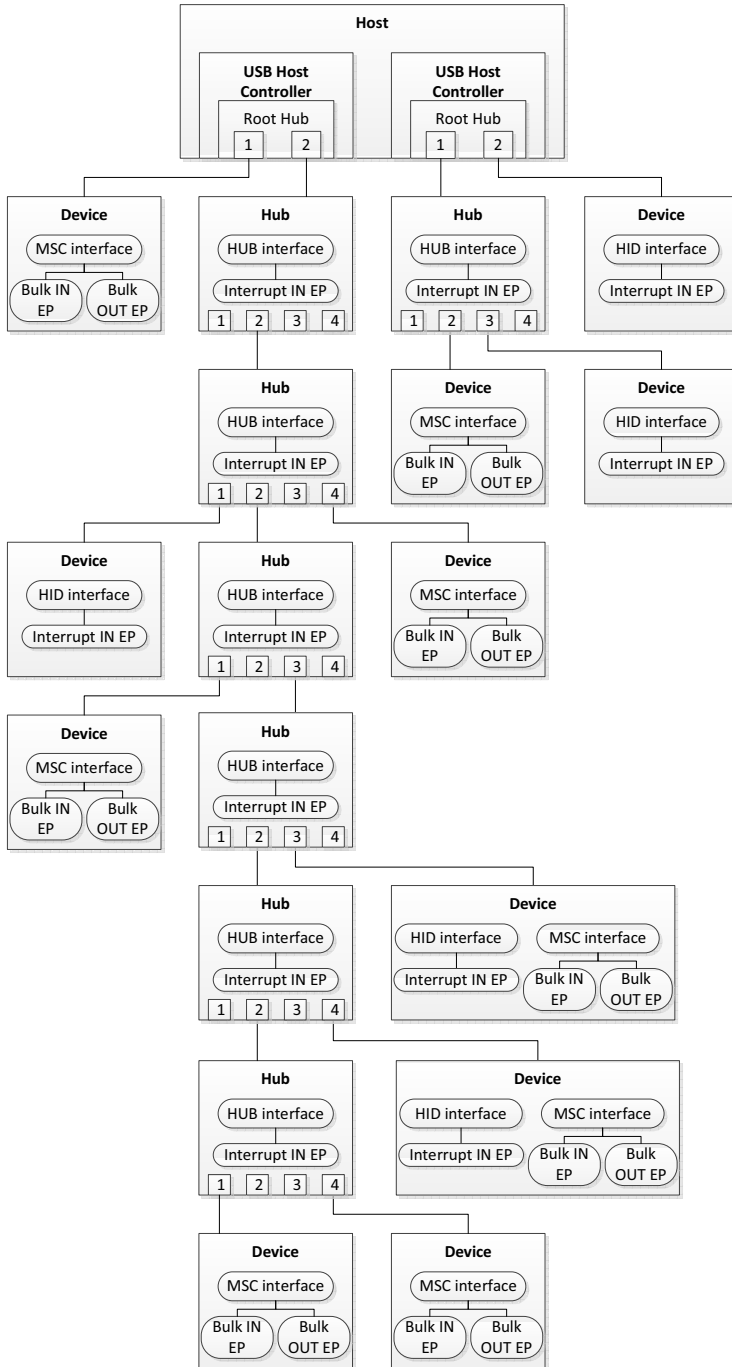


Figure 4-6 Multi-Host Controllers and Multiple Devices Configuration

Table 4-3 shows the values that should be set for the different configuration constants described earlier in order to support this bus topology.

<b>Configuration</b>	<b>Value</b>	<b>Explanation</b>
USBH_CFG_MAX_NBR_DEVS	17	6 external hubs + 11 other USB devices (in our example, 6 MSC, 3 HID and 2 composite devices). An external hub is also considered as a device. $\mu$ C/USB-Host enumerates an external hub as it would do with any other USB devices. Notice that if the maximum number of supported devices has been reached, the stack will simply disregard any other USB devices trying to connect. In that case, a message could be displayed to the user saying that the maximum number of devices has been reached.
USBH_CFG_MAX_NBR_CFGS	1	Only one configuration is usually needed for an external hub, a MSC device, a HID device or a composite device.
USBH_CFG_MAX_NBR_IFS	2	A single device with a Hub, MSC or HID function requires only one interface to be described. But the configuration inside the composite device is composed of two interfaces. Thus, we take the maximum of required interfaces among all configurations to be supported.
USBH_CFG_MAX_NBR_EPS	2	The maximum number of endpoints needed among all the interfaces in this bus topology is taken. MSC interface requires two bulk endpoints to communicate. Hub and HID interfaces use only one endpoint for communication.
USBH_CFG_MAX_NBR_CLASS_DRVS	3	Hub class (always present in the USB Host stack) + MSC class + HID class.

Configuration	Value	Explanation
USBH_CFG_MAX_CFG_DATA_LEN	100	<p>Size of a Configuration Descriptor sent by an external hub, a MSC or HID device must be less than 100 bytes.</p> <p>In general:</p> <p>A full HUB Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + size(Hub Descriptor) + 1* size(Endpoint Descriptor) = 9 + 9 + 9 + 1 * 7 = 34 bytes.</p> <p>A full MSC Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + 2 * size(Endpoint Descriptor) = 9 + 9 + 2 * 7 = 32 bytes. A full HID Configuration Descriptor = size(Configuration Descriptor) + size(Interface Descriptor) + size(HID Descriptor) + 1* size(Endpoint Descriptor) = 9 + 9 + 12 + 1 * 7 = 37 bytes.</p> <p><b>Full Configuration Descriptor for the composite device</b> = full HID Configuration Descriptor + size(Interface Descriptor for MSC) + 2 * size(Endpoint Descriptor) = 37 + 9 + 2 * 7 = <b>60 bytes</b>.</p> <p>Among HUB, MSC, HID and composite device, the composite device has the longest configuration descriptor. The value of <code>USBH_CFG_MAX_CFG_DATA_LEN</code> should be at least equal to the longest configuration descriptor size. A safety margin should be added. It is particularly true here because the HID descriptor has a size that varies according to the number of reports contained in the HID device. The HID descriptor size is given by this formula <math>(9 + (N * 3))</math> with N = number of reports.</p>
USBH_CFG_MAX_HUBS	8	2 root hubs + 6 external hubs.
USBH_CFG_MAX_HUB_PORTS	4	The number of active ports per hub is limited to 4. It limits also the number of other USB devices that the USB host stack can manage. In case 7-port hubs are used in this bus topology, only 4 devices will be managed by the USB host stack because the 3 other ports would be inactive. You may increase the value of this constant to 7 to allow up to 7 active ports per hub.
USBH_CFG_MAX_STR_LEN	256	This value ensures that the USB Host stack can get any length of strings of characters sent by any of USB devices. This value could be lower. In general, any string describing the device is less than 50 characters.
USBH_CFG_STD_REQ_TIMEOUT	5000	The device has 5 seconds to complete a standard request sent by the host.
USBH_CFG_STD_REQ_RETRY	3	If a standard request fails, the host attempts up to 3 times to get a successful request.
USBH_CFG_MAX_NBR_HC	2	Two host controllers on this platform.

---

<b>Configuration</b>	<b>Value</b>	<b>Explanation</b>
USBH_CFG_MAX_ISOC_DESC	1	Even if external hubs, MSC, HID, FTDI and CDC devices do not use isochronous endpoints, this value must be set to 1.
USBH_CFG_MAX_EXTRA_URB_PER_DEV	1	This value should be set at least to 1.

Table 4-3 Constant Values for Multi-Host Controllers and Multiple Devices Configuration Example

## Host Driver Guide

There are many USB Host Controllers (HC) available on the market and each requires a driver to work with  $\mu$ C/USB-Host. The amount of code necessary to port a specific USB HC to  $\mu$ C/USB-Host greatly depends on its complexity and features.

If not already available, a driver can be developed, as described in this chapter. However, it is recommended to modify an already existing host driver with the new host's specific code following the Micrium coding convention for consistency. It is also possible to adapt drivers written for other USB host stacks or examples provided by the chip manufacturer, especially if the driver is short and it is a matter of simply copying data to and from the HC.

This chapter describes the hardware (host) driver architecture for  $\mu$ C/USB-Host, including:

- Host Driver API Definition(s)
- Host Root Hub API Definition(s)
- Host Configuration
- Memory Allocation
- CPU and Board Support

Micrium provides sample configuration code free of charge; however, the sample code will likely require modifications depending on the combination of processor, evaluation board, and USB HC(s).



## 5-1 HOST DRIVER MODEL

No particular memory interface is required by  $\mu$ C/USB-Host's driver model. Therefore, the USB Host Controller (HC) may use the assistance of a Direct Memory Access (DMA) controller to transfer data or handle the data transfers directly.

## 5-2 HOST DRIVER API

### USB CONTROLLER API

All Host Controller Drivers (HCD) must declare an instance of the appropriate HCD API structure as a global variable within the source code. The API structure is an ordered list of function pointers utilized by  $\mu$ C/USB-Host when Host Controller (HC) hardware services are required.

A sample HCD API structure is shown in Listing 5-1.

```
USBH_HC_DRV_API USBH_<controller>_DrvAPI = {
    USBH_<controller>_Init,                (1)
    USBH_<controller>_Start,              (2)
    USBH_<controller>_Stop,               (3)
    USBH_<controller>_SpdGet,             (4)
    USBH_<controller>_Suspend,            (5)
    USBH_<controller>_Resume,             (6)
    USBH_<controller>_FrameNbrGet,        (7)

    USBH_<controller>_EP_Open,            (8)
    USBH_<controller>_EP_Close,           (9)
    USBH_<controller>_EP_Abort,           (10)
    USBH_<controller>_EP_IsHalt,          (11)

    USBH_<controller>_URB_Submit,         (12)
    USBH_<controller>_URB_Complete,       (13)
    USBH_<controller>_URB_Abort,         (14)
};
```

Listing 5-1 HCD Interface API

Note: It is the HCD developers' responsibility to ensure that all of the functions listed within the API are properly implemented and that the order of the functions within the API structure is correct. The different function pointers are:

- L5-1(1) HC initialization/add
- L5-1(2) HC start
- L5-1(3) HC stop
- L5-1(4) Get maximum speed of HC
- L5-1(5) HC suspend
- L5-1(6) HC resume
- L5-1(7) Retrieve frame number
- L5-1(8) Open an endpoint
- L5-1(9) Close an endpoint
- L5-1(10) Abort endpoint and pending URB(s)
- L5-1(11) Retrieve endpoint halt status
- L5-1(12) Submit an URB
- L5-1(13) Complete/free an URB
- L5-1(14) Abort an URB

The details of each API function are described in section F-1 "Host Driver Functions" on page 260.

## ROOT HUB API

All Host Controller Drivers (HCD) must declare an instance of the appropriate Root Hub (RH) API structure as a global variable within the source code. The RH API structure is an ordered list of function pointers utilized by  $\mu$ C/USB-Host when RH hardware services are required.

A sample HCD RH API structure is shown below.

```
USBH_HC_RH_API USBH_<controller>_RH_API = {
    USBH_<controller>_PortStatusGet,           (1)
    USBH_<controller>_HubDescGet,           (2)

    USBH_<controller>_PortEnSet,           (3)
    USBH_<controller>_PortEnClr,           (4)
    USBH_<controller>_PortEnChngClr,       (5)

    USBH_<controller>_PortPwrSet,          (6)
    USBH_<controller>_PortPwrClr,          (7)

    USBH_<controller>_PortResetSet,        (8)
    USBH_<controller>_PortResetChngClr,     (9)

    USBH_<controller>_PortSuspendClr,      (10)
    USBH_<controller>_PortConnChngClr,     (11)

    USBH_<controller>_RHSC_IntEn,          (12)
    USBH_<controller>_RHSC_IntDis,         (13)
};
```

Listing 5-2 RH Driver Interface API

The different function pointers are:

- L5-2(1) Get port status
- L5-2(2) Get hub descriptor
- L5-2(3) Set port enable
- L5-2(4) Clear port enable
- L5-2(5) Clear port enable change flag
- L5-2(6) Set power on port

- L5-2(7)      Clear power on port
- L5-2(8)      Set reset state on port
- L5-2(9)      Clear port reset change flag
- L5-2(10)     Clear port suspend
- L5-2(11)     Clear port connection change flag
- L5-2(12)     Enable RH interrupts
- L5-2(13)     Disable RH interrupts

The details of each RH API function are described in Appendix F, “Root Hub Driver Functions” on page 278.

Note:  $\mu$ C/USB-Host HCD API function names may not be unique. Name clashes between HCDs are avoided by never globally prototyping HCD functions and ensuring that all references to functions within the driver are obtained by pointers within the API structure. The developer may arbitrarily name the functions within the source file so long as the API structure is properly declared. The user application should never need to call API functions. Unless special care is taken, calling HCD functions may lead to unpredictable results due to reentrancy.

When writing your own HCD, you can assume that each driver API function accepts a pointer to a structure of the type `USBH_HC_DRV` as its first parameter. Through this structure, you will be able to access the following fields:

```
typedef struct usbh_hc_drv USBH_HC_DRV;

struct usbh_hc_drv {
    CPU_INT08U      Nbr;                (1)
    void            *DataPtr;           (2)
    USBH_DEV        *RH_DevPtr;        (3)
    USBH_HC_CFG     *HC_CfgPtr;        (4)
    USBH_HC_DRV_API *API_Ptr;          (5)
    USBH_HC_RH_API  *RH_API_Ptr;       (6)
    USBH_HC_BSP_API *BSP_API_Ptr;      (7)
};
```

Listing 5-3 **USB HC Driver Data Type**

- L5-3(1) Unique index to identify HC.
- L5-3(2) Pointer to HCD specific data.
- L5-3(3) Pointer to RH device structure.
- L5-3(4) Pointer to HCD configuration.
- L5-3(5) Pointer to HCD API structure.
- L5-3(6) Pointer to RH API structure.
- L5-3(7) Pointer to Board Support Package (BSP) API structure.

### **5-3 INTERRUPT HANDLING**

Interrupt handling is accomplished using the following multi-level scheme.

- 1 Processor level kernel-aware interrupt handler
- 2 Host Controller Driver (HCD) interrupt handler

During initialization, the HCD registers all necessary interrupt sources with the Board Support Package (BSP) interrupt management code. You can also accomplish this by plugging an interrupt vector table during compile time. Once the global interrupt vector sources are configured and an interrupt occurs, the system will call the first-level interrupt handler. The first-level interrupt handler is responsible for performing all kernel required steps prior to calling the USB HCD interrupt handler: `USBH_<controller>_ISR_Handler()`. Depending on the platform architecture (that is the way the kernel handles interrupts) and the USB HC interrupt vectors, the HCD interrupt handler implementation may follow one of the models described in the next sections.

### **5-3-1 SINGLE USB ISR VECTOR WITH ISR HANDLER ARGUMENT**

If the platform architecture allows parameters to be passed to ISR handlers and the USB HC has a single interrupt vector for the USB host, the first-level interrupt handler may be defined as:

#### **PROTOTYPE**

```
void USBH_<controller>_BSP_IntHandler (void *p_arg);
```

#### **ARGUMENTS**

**p\_arg**            Pointer to USB HC driver structure that must be typecast to a pointer to USBH\_HC\_DRV.

### **5-3-2 SINGLE USB ISR VECTOR**

If the platform architecture does not allow parameters to be passed to ISR handlers and the USB HC has a single interrupt vector for the USB host, the first-level interrupt handler may be defined as:

#### **PROTOTYPE**

```
void USBH_<controller>_BSP_IntHandler (void);
```

#### **ARGUMENTS**

None.

#### **NOTES / WARNINGS**

In this configuration, the pointer to the USB HC driver structure must be stored globally in the BSP. Since the pointer to the USB HC driver structure is never modified, the BSP initialization function, `USBH_<controller>_BSP_Init()`, can save its address for later use.

### **5-3-3 MULTIPLE USB ISR VECTORS WITH ISR HANDLER ARGUMENTS**

If the platform architecture allows parameters to be passed to ISR handlers and the USB HC has multiple interrupt vectors for the USB host (e.g., USB events, DMA transfers, ...), the first-level interrupt handler may need to be split into multiple sub-handlers. Each sub-handler would be responsible for managing the status reported to the different vectors. For example, the first-level interrupt handlers for a USB HC that redirects USB events to one interrupt vector and the status of DMA transfers to a second interrupt vector may be defined as:

#### **PROTOTYPE**

```
void USBH_<controller>_BSP_EventIntHandler (void *p_arg);  
void USBH_<controller>_BSP_DMA_IntHandler (void *p_arg);
```

#### **ARGUMENTS**

`p_arg`            Pointer to USB HC driver structure that must be typecast to a pointer to `USBH_HC_DRV`.

### **5-3-4 MULTIPLE USB ISR VECTORS**

If the platform architecture does not allow parameters to be passed to ISR handlers and the USB HC has multiple interrupt vectors for the USB host (e.g., USB events, DMA transfers), the first-level interrupt handler may need to be split into multiple sub-handlers. Each sub-handler would be responsible for managing the status reported to the different vectors. For example, the first-level interrupt handlers for a USB HC that redirects USB events to one interrupt vector and the status of DMA transfers to a second interrupt vector may be defined as:

#### **PROTOTYPE**

```
void USBH_<controller>_BSP_EventIntHandler (void);  
void USBH_<controller>_BSP_DMA_IntHandler (void);
```

#### **ARGUMENTS**

None.

## NOTES / WARNINGS

In this configuration, the pointer to the USB HC driver structure must be stored globally in the BSP. Since the pointer to the USB HC driver structure is never modified, the BSP initialization function, `USBH_<controller>_BSP_Init()`, can save its address for later use.

## 5-4 HOST CONTROLLER DRIVER CONFIGURATION

The USB Host Controller Driver (HCD) characteristics must be shared with the USB host stack through configuration parameters. All of these parameters are provided through a global structure of type `USBH_HC_CFG`. This structure is declared in the file `usbh_hc_cfg.h`, and defined in the file `usbh_hc_cfg.c` (refer to section “Modify Host Controller Configuration” on page 29 for an example on how to initialize this structure). These files are distributed as templates, and you should modify them to have the proper configuration for your USB Host Controller (HC). Refer to section 4-3 “Host Controller Driver Configuration” on page 56 for further details on HCD configuration.

## 5-5 MEMORY ALLOCATION

Memory allocation in the driver can be simplified by the use of memory allocation functions available from Micrium’s `μC/LIB` module. `μC/LIB`’s memory allocation functions provide allocation of memory from dedicated memory space (e.g., USB RAM) or general purpose heap. The driver may use the pool functionality offered by `μC/LIB`. Memory pools use fixed-sized blocks that can be dynamically allocated and freed during application execution. Memory pools may be convenient to manage objects needed by the driver. The objects could be for instance data structures mandatory for DMA operations. For more information on using `μC/LIB` memory allocation functions, consult the `μC/LIB` documentation.



## 5-6 CPU AND BOARD SUPPORT

The USB host stack supports big-endian and little-endian CPU architectures.

In order for HCDs to be platform-independent, it is necessary to provide a layer of code that abstracts details such as clocks, interrupt controllers, input/output (IO) pins, and other hardware modules configuration. With this board support package (BSP) code layer, it is possible for the majority of the USB host stack to be independent of any specific hardware, and for device drivers to be reused on different architectures and bus configurations without the need to modify stack or driver source code. These procedures are also referred as the USB host BSP for a particular development board.

A sample host BSP interface API structure is shown in Listing 5-4.

```
USBH_HC_BSP_API USBH_<controller>_BSP_API = {
    USBH_<controller>_BSP_Init,           (1)
    USBH_<controller>_BSP_IntReg,        (2)
    USBH_<controller>_BSP_IntUnReg      (3)
};
```

Listing 5-4 Host Controller BSP Interface API

- L5-4(1)     HC BSP initialization function pointer
- L5-4(2)     HC BSP interrupt register function pointer
- L5-4(3)     HC BSP interrupt un-register function pointer

The details of each HC BSP API function are described in section F-3 “Host Driver BSP Functions” on page 292.

---

## 5-7 USB HOST CONTROLLER DRIVER FUNCTIONAL MODEL

This section describes the functional model of some typical operations in which the HCD will be involved. These functional models include:

- Root Hub interactions
- Endpoint open
- URB submit

### 5-7-1 ROOT HUB INTERACTIONS

Typically, a Root Hub (RH) will be addressed via a (set) of hardware registers. From the point of view of the core module, the RH is not different from another external hub. It will be managed by the Hub class through the use of Hub class-specific requests. It must then provide information on RH port(s) in the same format as would do an external hub using the *GetPortStatus* request. This request returns two fields: *wPortStatus* and *wPortChange*. Some Host Controller (HC) hardware will provide port(s) status in this format via their hardware registers, but some will not. In case they don't, the driver should declare the two following variables for each of its RH port(s) within its internal data structure as shown in Listing 5-5.

```
typedef struct  usbh_<controller>_data {
    ...
    CPU_INT16U  RH_PortStatus[<port number>];
    CPU_INT16U  RH_PortChng[<port number>];
    ...
} USBH_<controller>_DATA;
```

Listing 5-5 Root Hub Port Status Variables

The variable `RH_PortStatus[<port_number>]` should be filled the same way as the *wPortStatus* field of the *GetPortStatus* request. See Table 11-21 of the *Universal Serial Bus Specification, revision 2.0*, at page 427 for more details.

The variable `RH_PortChng[<port_number>]` should be filled the same way as the `wPortChange` field of the `GetPortStatus` request. See Table 11-22 of the *Universal Serial Bus Specification, revision 2.0*, at page 431 for more details.

These fields will be requested by the core using the `USBH_<controller>_PortStatusGet()` function. Depending on your Host Controller (HC) hardware, some of the bits can be updated by reading hardware registers when this function is invoked or when interrupts are triggered. When your hardware controller generates an interrupt to inform about a device connect/disconnect, it is also important to notify the core module after updating the `RH_PortStatus` and `RH_PortChng` variables (if necessary). This is done by calling `USBH_HUB_RH_Event(p_hc_drv->RH_DevPtr)`. Listing 5-6 shows an example of how the core should be notified of a device connect/disconnect from the ISR.

```
if (<device connected>) {
    DEF_BIT_SET(p_data->RH_PortChng,    USBH_HUB_STATUS_PORT_CONN);
    DEF_BIT_SET(p_data->RH_PortStatus,  USBH_HUB_STATUS_PORT_CONN);

    USBH_HUB_RH_Event(p_hc_drv->RH_DevPtr);
}

if (<device disconnected>) {
    DEF_BIT_SET(p_data->RH_PortChng,    USBH_HUB_STATUS_PORT_CONN);
    DEF_BIT_CLR(p_data->RH_PortStatus,  USBH_HUB_STATUS_PORT_CONN);

    USBH_HUB_RH_Event(p_hc_drv->RH_DevPtr);
}
```

Listing 5-6 **Device Connect/Disconnect Core Notification**

## 5-7-2 ENDPOINT OPENING

### LIST-BASED HOST CONTROLLERS

List-based controllers (like OHCI or EHCI controllers) generally have two or more lists of opened endpoints (*async* list, *periodic* list, ...). The lists represent data structures that are located in a memory region that is shared between the HC and HCD. Depending on the format of these lists and on the type of endpoint, opening an endpoint generally simply consist in allocating, initializing and adding a new entry to the list for the endpoint. Bandwidth usage should also be updated when a new periodic endpoint (interrupt and isochronous) is added.

## NON LIST-BASED HOST CONTROLLERS

Host Controllers (HC) that are not list-based generally have a limited list of pipes used as endpoints. Moreover, the pipes may also have a defined type (control, bulk, interrupt or isochronous). It is the responsibility of the `USBH_<controller>_EP_Open()` function to find an available pipe of the requested type and mark it as used.

### 5-7-3 URB SUBMIT

A USB Request Block (URB) is a structure that represents a USB transfer and that contains important information on that transfer. In  $\mu$ C/USB-Host, an URB is represented using the `USBH_URB` structure. Some of the fields of this structure must be set/updated or read by the driver. Listing 5-7 presents some important fields of the `USBH_URB` structure.

```
typedef struct usbh_urb USBH_URB;

struct usbh_urb {
    CPU_REG08      State;                (1)
    USBH_EP        *EP_Ptr;              (2)
    USBH_ERR        Err;                 (3)

    void           *UserBufPtr;          (4)
    CPU_INT32U     UserBufLen;           (5)
    void           *DMA_BufPtr;          (6)
    CPU_INT32U     DMA_BufLen;           (7)
    CPU_INT32U     XferLen;              (8)

    void           *ArgPtr;              (9)

    USBH_TOKEN     Token;                (10)

    ...
};
```

Listing 5-7 **USBH\_URB Structure Important Fields**

L5-7(1)      Contains the state of the URB. The driver should not set this field but might need to read it. The available URB states are described in Table 5-1.

<b>Error code</b>	<b>Description</b>
USBH_URB_STATE_NONE	The URB has been submitted to the core.
USBH_URB_STATE_SCHEDULED	The URB has been scheduled in the HC.
USBH_URB_STATE_QUEUED	The URB has completed.
USBH_URB_STATE_ABORTED	The URB has been aborted by the core or the application.

Table 5-1 **URB States**

L5-7(2) Pointer to the endpoint structure to which this URB is attached. Should not be modified by the driver.

L5-7(3) Field that should be set by the driver to inform the core of any error during transfer. Table 5-2 summarizes some error codes that a driver could assign to this field.

<b>Error code</b>	<b>Description</b>
USBH_ERR_NONE	URB has completed successfully.
USBH_ERR_HC_IO	Input Output (IO) error occurred with the HC.
USBH_ERR_EP_STALL	Endpoint is in a stall condition.
USBH_ERR_EP_NACK	Device returned a NACK handshake and the HC has no hardware assistance to re-submit the URB.

Table 5-2 **URB Error Codes**

L5-7(4) Pointer to the data buffer that needs to be transferred/filled with received data. Should not be modified by the driver.

L5-7(5) Length, in octets, of the data buffer. Should not be modified by the driver.

L5-7(6) On DMA-based controllers, pointer to the data buffer that will be transferred/filled with received data by the DMA controller. Especially useful when the buffer needs to be copied into a USB dedicated memory to be read/written by the DMA controller.

- L5-7(7) Length, in octets, of the DMA buffer.
- L5-7(8) Contains the length, in octets, of the data transferred. This must be updated by the driver.
- L5-7(9) Pointer available to set driver specific data related to this URB.
- L5-7(10) Indicate the token of the transfer. Especially useful with control transfers. This field should not be modified by the driver. Can be one of the following:
- `USBH_TOKEN_SETUP`
  - `USBH_TOKEN_OUT`
  - `USBH_TOKEN_IN`

### **LIST-BASED HOST CONTROLLERS WITHOUT DEDICATED MEMORY**

On a list-based Host Controller (HC), submitting a new URB generally consists in allocating, initializing and adding a transfer to the endpoint list. The DMA controller will access the buffer data directly from the application buffer. Once the HC notifies of the transfer completion via an interrupt, the driver must update the `p_urb->XferLen` field accordingly and call `USBH_URB_Done()` within its ISR.

### **LIST-BASED HOST CONTROLLERS WITH DEDICATED MEMORY**

Submitting an URB on a list-based controller using dedicated memory for data buffer is similar to a controller that does not use dedicated memory. For OUT endpoints, the main difference is that before inserting the transfer to the endpoint list, the driver must copy the buffer content to the dedicated memory. The field `p_urb->DMA_BufPtr` should be used by the driver to point to the buffer in dedicated memory. For IN endpoints, before submitting a transfer, the driver must set the field `p_urb->DMA_BufPtr` to make it point to an empty region of the dedicated memory that can hold up to `p_urb->UserBufLen` octets of data. Once the transfer has completed and the function `USBH_URB_Done()` has been invoked, the core module will call the function `USBH_<controller>_URB_Complete()`. The driver must copy the data to the field `p_urb->UserBufPtr` at this moment.

## NON LIST-BASED HOST CONTROLLERS

Host Controllers (HC) that are not list-based generally use a FIFO for data transfers. FIFOs generally have a limited depth and are likely to be unable to hold an entire transfer, especially if it is a large transfer. Figure 5-1 summarizes the operations that must be done by the driver in this case.

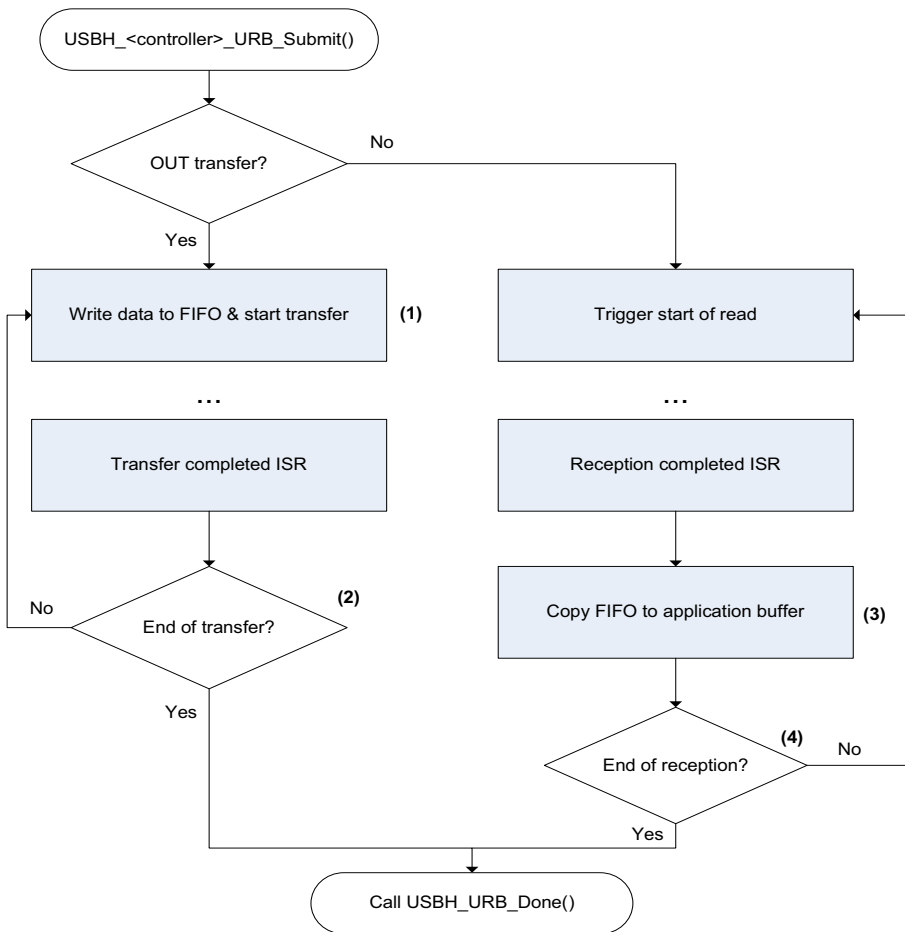


Figure 5-1 URB Submit when HC is not List-Based

F5-1(1) Driver must write data to FIFO until it is full or the buffer is empty.

- F5-1(2) The driver can now update the `p_urb->XferLen` field. It must then determine if the transfer is completed. It can simply be determined by comparing the total amount of data transferred with `p_urb->UserBufLen`.
- F5-1(3) Once the reception has completed, the driver must copy the data from the FIFO to the buffer at `p_urb->UserBufPtr`.
- F5-1(4) Many conditions must be considered to evaluate if a reception is complete. The following list describes the condition where the reception is considered complete.
- The last transaction had a length of zero.
  - The last transaction had a length that is less than the maximum packet size of the endpoint.
  - The total amount of data received is equal to `p_urb->UserBufLen`.





## Communication Device Class

This chapter describes the Communications Device Class (CDC) class and the associated CDC subclass supported by  $\mu$ C/USB-Host.  $\mu$ C/USB-Host currently supports the Abstract Control Model (ACM) subclass.

The CDC and the associated subclass implementation complies with the following specifications:

- *Universal Serial Bus, Class Definitions for Communications Devices, revision 1.2*, November 3 2010.
- *Universal Serial Bus, Communications, Subclass for PSTN Devices, revision 1.2*, February 9, 2007.

CDC includes various telecommunication and networking devices. Telecommunication devices encompass analog modems, analog and digital telephones, ISDN terminal adapters, etc. Networking devices contain, for example, ADSL and cable modems, Ethernet adapters and hubs. CDC defines a framework to encapsulate existing communication services standards, such as V.250 (for modems over telephone network) and Ethernet (for local area network devices), using a USB link. A communication device is in charge of device management, call management when needed and data transmission. CDC defines seven major groups of devices. Each group belongs to a model of communication which may include several subclasses. Each group of devices has its own specification document besides the CDC base class. The seven groups are:

- Public Switched Telephone Network (PSTN), devices including voiceband modems, telephones and serial emulation devices.
- Integrated Services Digital Network (ISDN) devices, including terminal adaptors and telephones.

- Ethernet Control Model (ECM) devices, including devices supporting the IEEE 802 family (for instance cable and ADSL modems, WiFi adaptors).
- Asynchronous Transfer Mode (ATM) devices, including ADSL modems and other devices connected to ATM networks (workstations, routers, LAN switches).
- Wireless Mobile Communications (WMC) devices, including multi-function communications handset devices used to manage voice and data communications.
- Ethernet Emulation Model (EEM) devices which exchange Ethernet-framed data.
- Network Control Model (NCM) devices, including high-speed network devices (High Speed Packet Access modems, Line Terminal Equipment)

## **6-1 OVERVIEW**

A CDC device is composed of several interfaces to implement a certain function. It is formed by the following interfaces:

- Communications Class Interface (CCI)
- Data Class Interface (DCI)

A CCI is responsible for the device management and optionally the call management. The device management enables the general configuration and control of the device and the notification of events to the host. The call management enables the establishment and termination of calls. Call management might be multiplexed through a DCI. A CCI is mandatory for all CDC devices. It identifies the CDC function by specifying the communication model supported by the CDC device. The interface(s) following the CCI can be any defined USB class interface, such as Audio or a vendor-specific interface. The vendor-specific interface is represented specifically by a DCI.

A DCI is responsible for data transmission. The data transmitted and/or received do not follow a specific format. Data could be raw data from a communication line, data following a proprietary format, etc. All the DCIs following the CCI can be seen as subordinate interfaces.

A CDC device must have at least one CCI and zero or more DCIs. One CCI and any subordinate DCI together provide a feature to the host. This capability is also referred to as a function. In a CDC composite device, you could have several functions. And thus, the device would be composed of several sets of CCI and DCI(s) as shown in Figure 6-1.

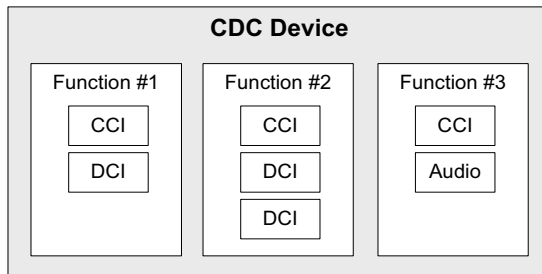


Figure 6-1 CDC Composite Device

A CDC device is likely to use the following combination of endpoints:

- A pair of control IN and OUT endpoints called the default endpoint.
- An optional bulk or interrupt IN endpoint.
- A pair of bulk or isochronous IN and OUT endpoints.

Table 6-1 provides a list of all the different types of endpoints distinguished by their data flow direction, interface and application.

Endpoint	Direction	Interface	Use for
Control IN	Device-to-host	CCI	Standard requests for enumeration, class-specific requests, device management and optionally call management.
Control OUT	Host-to-device	CCI	Standard requests for enumeration, class-specific requests, device management and optionally call management.
Interrupt or bulk IN	Device-to-host	CCI	Events notification, such as ring detect, serial line status, network status.
Bulk or isochronous IN	Device-to-host	DCI	Raw or formatted data communication.
Bulk or isochronous OUT	Host-to-device	DCI	Raw or formatted data communication.

Table 6-1 CDC Endpoint Use

Most communication devices use an interrupt endpoint to notify the host of any events.

The seven major models of communication encompass several subclasses. A subclass describes the way the device should use the CCI to handle the device management and call management. Table 6-2 shows all the possible subclasses and the communication model they belong to.

<b>Subclass</b>	<b>Communication model</b>	<b>Example of devices using this subclass</b>
Direct Line Control Model	PSTN	Modem devices directly controlled by the USB host
Abstract Control Model	PSTN	Serial emulation devices, modem devices controlled through a serial command set
Telephone Control Model	PSTN	Voice telephony devices
Multi-Channel Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
CAPI Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
Ethernet Networking Control Model	ECM	DOC-SIS cable modems, ADSL modems that support PPPoE emulation, Wi-Fi adaptors (IEEE 802.11-family), IEEE 802.3 adaptors
ATM Networking Control Model	ATM	ADSL modems
Wireless Handset Control Model	WMC	Mobile terminal equipment connecting to wireless devices
Device Management	WMC	Mobile terminal equipment connecting to wireless devices
Mobile Direct Line Model	WMC	Mobile terminal equipment connecting to wireless devices
OBEX	WMC	Mobile terminal equipment connecting to wireless devices
Ethernet Emulation Model	EEM	Devices using Ethernet frames as the next layer of transport. Not intended for routing and Internet connectivity devices
Network Control Model	NCM	IEEE 802.3 adaptors carrying high-speed data bandwidth on network

Table 6-2 **CDC Subclasses**

## 6-2 CLASS IMPLEMENTATION

$\mu$ C/USB-Host Communication Device Class (CDC) is designed to allow subclasses to be built on top of it. It offers an API to the subclasses that allows to communicate using Data Class Interface (DCI) and Communication Class Interface (CCI). Figure 6-2 shows how CDC base and subclass(es) interact.

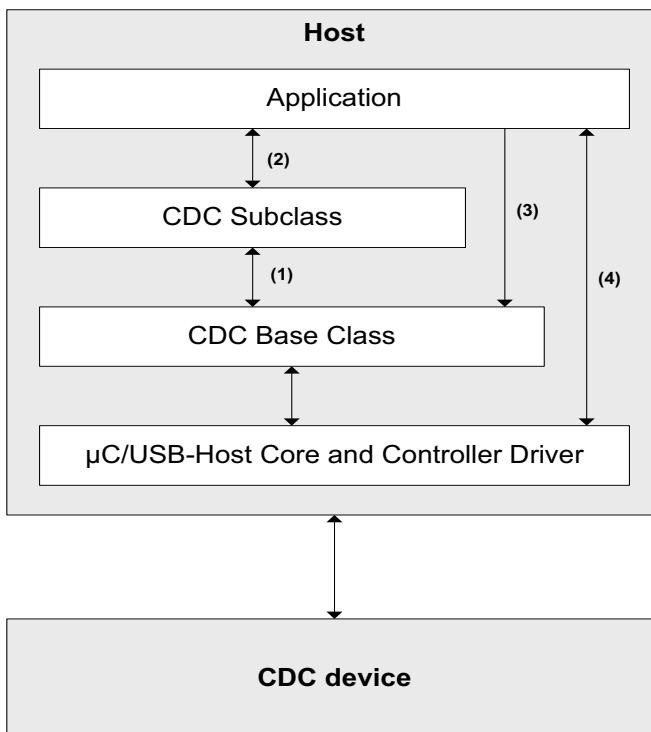


Figure 6-2 **CDC Implementation**

- F6-2(1) CDC base class offers an API to the subclasses that allows to communicate with a CDC device via DCI and CCI.
- F6-2(2) Your application interacts with the subclass for the communication.
- F6-2(3) Upon device connection, your application must add a reference on the device and release it on device disconnection.
- F6-2(4) At initialization, your application must register the CDC driver to the core.

## 6-3 CONFIGURATION AND INITIALIZATION

### 6-3-1 GENERAL CONFIGURATION

There is only one configuration constant necessary to customize the CDC host base class. This constant is located in the `usbh_cfg.h` file. Table 6-3 shows a description of this constant.

Constant	Description
<code>USBH_CDC_CFG_MAX_DEV</code>	Configures the maximum number of CDC functions the class can handle.

Table 6-3 CDC Configuration Constant

### 6-3-2 CLASS INITIALIZATION

In order to be integrated to the core and considered on a device connection, the CDC base class driver must be added to the core class driver list. This is done by calling `USBH_ClassDrvReg()` and is described in Listing 6-1.

```

USBH_ERR App_USBH_CDC_Init (void)
{
    USBH_ERR    err;

    ...

    err = USBH_ClassDrvReg(    &USBH_CDC_ClassDrv,           (1)
                              App_USBH_CDC_ClassNotify,      (2)
                              (void *)0);                    (3)

    return(err);
}

```

Listing 6-1 CDC Initialization

- L6-1(1) First parameter is a CDC class driver structure. It is defined in `usbh_cdc.h`.
- L6-1(2) Second parameter is a pointer to the application's callback function. This function will be called upon CDC device connection/disconnection.
- L6-1(3) Last parameter is an optional pointer to application specific data.

### 6-3-3 DEVICE CONNECTION AND DISCONNECTION HANDLING

Upon connection/disconnection of a CDC device, your application will be notified via a callback function. Listing describes the operations that must be performed at that moment.

```

static void App_USBH_CDC_ClassNotify (void      *p_class_dev,
                                      CPU_INT08U is_conn,
                                      void      *p_ctx)
{
    USBH_ERR      err;
    USBH_CDC_DEV *p_cdc_dev;
    CPU_INT08U    sub_class;
    CPU_INT08U    protocol;

    (void)&p_ctx;
    p_cdc_dev = (USBH_CDC_DEV *)p_class_dev;
    switch (is_conn) {
        case USBH_CLASS_DEV_STATE_CONN:
            err = USBH_CDC_RefAdd(p_cdc_dev);           (1)
            if (err != USBH_ERR_NONE) {
                /* $$$ Handle error */
                return;
            }

            err = USBH_CDC_SubclassGet(p_cdc_dev, &sub_class); (2)
            if (err != USBH_ERR_NONE) {
                /* $$$ Handle error */
                (void)USBH_CDC_RefRel(p_cdc_dev);
                return;
            }

            err = USBH_CDC_ProtocolGet(p_cdc_dev, &protocol); (3)
            if (err != USBH_ERR_NONE) {
                /* $$$ Handle error */
                (void)USBH_CDC_RefRel(p_cdc_dev);
                return;
            }

            /* $$$ Subclass specific operations. */ (4)
            break;
    }
}

```



```
case USBH_CLASS_DEV_STATE_DISCONN:

    /* $$$ Subclass specific operations. */ (5)
    (void)USBH_CDC_RefRel(p_cdc_dev); (6)
    break;

default:
    break;
}
}
```

Listing 6-2 CDC Device Connection/Disconnection

- L6-2(1) Add an application reference to this CDC device by calling `USBH_CDC_RefAdd()`.
- L6-2(2) At this point, your application is not aware of what is the CDC subclass used by the device. A call to this function will let you retrieve the subclass code.
- L6-2(3) Your application can optionally need to retrieve the protocol code of the CDC device.
- L6-2(4) Subclass specific initialization must be performed here. Refer to your specific subclass section of this chapter for more details.
- L6-2(5) Subclass specific de-initialization must be performed here. Refer to your specific subclass section of this chapter for more details.
- L6-2(6) On a CDC device disconnection, your application must release its reference.

## 6-4 ABSTRACT CONTROL MODEL (ACM) SUBCLASS

The ACM subclass is used by two types of communication devices:

- Devices supporting AT commands (for instance, voiceband modems).
- Serial emulation devices which are also called Virtual COM port devices.

Micrium's ACM subclass implementation complies with the following specification:

*Universal Serial Bus, Communications, Subclass for PSTN Devices, revision 1.2, February 9, 2007.*

### 6-4-1 CONFIGURATION AND INITIALIZATION

#### GENERAL CONFIGURATION

There is only one configuration constant necessary to customize the ACM subclass. This constant is located in the `usbh_cfg.h` file. Table 6-3 shows a description of this constant.

Constant	Description
<code>USBH_CDC_ACM_CFG_MAX_DEV</code>	Configures the maximum number of CDC ACM functions the class can handle. Should be $\leq$ <code>USBH_CDC_CFG_MAX_DEV</code> .

Table 6-4 CDC ACM Subclass Configuration Constant

#### SUBCLASS INITIALIZATION

The ACM subclass does not have to be registered to the core, but it must be initialized. This is done by calling `USBH_CDC_ACM_GlobalInit()`.

## 6-4-2 CONNECTION AND DISCONNECTION HANDLING

Upon connection/disconnection of a CDC ACM device, your application will be notified via a callback function as described in Listing . A few ACM subclass specific steps must be performed as well at that moment.

ACM subclass specific operations on device connection and disconnection are described in Listing 6-3.

```

static void App_USBH_CDC_ClassNotify (USBH_CDC_DEV *p_class_dev,
                                      CPU_INT08U   is_conn,
                                      void          *p_ctx)
{
    USBH_ERR      err;
    USBH_CDC_ACM_DEV *p_cdc_acm_dev;
    CPU_INT08U    sub_class;
    CPU_INT08U    protocol;

    (void)&p_ctx;

    switch (is_conn) {
        case USBH_CLASS_DEV_STATE_CONN:

            /* $$$$ CDC base class specific operations. */

            if (sub_class != USBH_CDC_CONTROL_SUBCLASS_CODE_ACM) {
                /* $$$$ Handle error */
                (void)USBH_CDC_RefRel(p_cdc_dev);
                return;
            }

            if ((protocol != USBH_CDC_CONTROL_PROTOCOL_CODE_USB ) &&
                (protocol != USBH_CDC_CONTROL_PROTOCOL_CODE_V_25_AT) &&
                (protocol != USBH_CDC_CONTROL_PROTOCOL_CODE_VENDOR )) {
                /* $$$$ Handle error */
                (void)USBH_CDC_RefRel(p_cdc_dev);
                return;
            }
    }
}

```

```

p_cdc_acm_dev = USBH_CDC_ACM_Add(p_cdc_dev,                               (3)
                                &err);

if (err != USBH_ERR_NONE) {
    /* $$$$ Handle error */
    (void)USBH_CDC_RefRel(p_cdc_dev);
    return;
}

USBH_CDC_ACM_EventRxNotifyReg(p_cdc_acm_dev,                            (4)
                              App_USBH_CDC_ACM_SerEventNotify);

err = USBH_CDC_ACM_LineCodingSet(p_cdc_acm_dev,                        (5)
                                 USBH_CDC_ACM_LINE_CODING_BAUDRATE_19200,
                                 USBH_CDC_ACM_LINE_CODING_STOP_BIT_2,
                                 USBH_CDC_ACM_LINE_CODING_PARITY_NONE,
                                 USBH_CDC_ACM_LINE_CODING_DATA_BITS_8);

if (err != USBH_ERR_NONE) {
    /* $$$$ Handle error */
    (void)USBH_CDC_ACM_Remove(p_cdc_acm_dev);
    (void)USBH_CDC_RefRel(p_cdc_dev);
    return;
}

err = USBH_CDC_ACM_LineStateSet(p_cdc_acm_dev,                        (6)
                                USBH_CDC_ACM_DTR_SET,
                                USBH_CDC_ACM_RTS_SET);

if (err != USBH_ERR_NONE) {
    /* $$$$ Handle error */
    (void)USBH_CDC_ACM_Remove(p_cdc_acm_dev);
    (void)USBH_CDC_RefRel(p_cdc_dev);
    return;
}
break;

case USBH_CLASS_DEV_STATE_DISCONN:
    USBH_CDC_ACM_Remove(p_cdc_acm_dev);                                (7)

    /* $$$$ CDC base class specific operations. */
    break;

default:
    break;
}
}

```

Listing 6-3 CDC ACM Device Connection/Disconnection

- L6-3(1) Ensure that the connected CDC device uses ACM subclass.
- L6-3(2) Make sure the protocol code used is supported by the subclass/application.
- L6-3(3) Instantiate an ACM device for the connected CDC device.
- L6-3(4) Registers an application callback function that will be called by the subclass on reception of notification event from the device.
- L6-3(5) Set device's line coding parameters.
- L6-3(6) Set device line state.
- L6-3(7) Free the instantiated ACM reference on device disconnection.

### **6-4-3 DEMO APPLICATION**

A simple demo application is provided with the host Abstract Control Model (ACM) subclass. This demo application can be used as a starting point to create your own application. It is intended to be used with a CDC ACM USB modem that supports AT commands. For more information on AT commands, visit the following web page: <http://support.microsoft.com/kb/164660>. Note that the demo application handles only one CDC ACM device at a time and thus you should not attempt to connect more than one CDC ACM device at a time.

Upon a CDC ACM device connection, the demo application will initiate a few AT commands and wait for modem's response. Here is the list of the commands that will be issued:

- ATQ0V1E0
- ATIO
- AT11
- AT12
- AT13
- AT17

The response received from the device will then be displayed on the output terminal if application tracing is enabled. Listing 6-4 shows an example of output given by the demo application with an *USRobotics Model 5637* modem.

```
===== AT CMD =====
ATQ0V1E0
===== STATUS / DATA =====
ATQ0V1E0
OK

===== AT CMD =====
ATI0
===== STATUS / DATA =====

5601

OK

===== AT CMD =====
ATI1
===== STATUS / DATA =====

5588

OK

===== AT CMD =====
ATI2
===== STATUS / DATA =====

OK

===== AT CMD =====
ATI3
===== STATUS / DATA =====

U.S. Robotics 56K FAX USB V1.2.23

OK

===== AT CMD =====
ATI7
===== STATUS / DATA =====

Configuration Profile...

Product Type          Canada USB
...
```

Listing 6-4 CDC ACM Demo Application Output Example

---

## DEMO APPLICATION CONFIGURATION

Before running the CDC ACM demo application, you must configure it properly. The configuration constants that must be set are located in the `app_cfg.h` file. Table 6-5 lists the preprocessor constants that must be defined.

Preprocessor Constants	Description	Default Value
<code>APP_CFG_USBH_EN</code>	Enables $\mu$ C/USB-Host in your application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_CDC_EN</code>	Enables CDC ACM demo application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_CDC_MAIN_TASK_Prio</code>	CDC ACM application task priority.	21
<code>APP_CFG_USBH_CDC_MAIN_TASK_STK_SIZE</code>	CDC ACM application task stack size.	1000

Table 6-5 Demo Application Configuration Constants

## Future Technology Devices International Class

This chapter describes the Future Technology Devices International (FTDI) class supported by  $\mu$ C/USB-Host. The FTDI implementation complies with the following specification:

- *API for FTxxxx Devices Application Note AN\_115*, 4/04/11, Version 1.1.<sup>1</sup>

USB-to-serial converters enable USB based systems to access and communicate with serial devices featuring D-Sub. RS-232 devices are still found mainly in industrial machines or scientific instruments but the USB standard has replaced the serial port in many applications. To solve this problem, the FTDI chips offer USB migration by combining USB-Serial (USB-RS232) and USB-FIFO silicon solutions. The FTDI class implements the basic vendor requests in order to set the connection and transmit data using bulk endpoints.

$\mu$ C/USB-Host's FTDI class implementation supports the following chips:

- FT8U232AM
- FT232B/FT232R/FT232H
- FT2232D/FT2232H
- FT4232H

---

1.The reference document can be requested from Future Technology Devices International Ltd. after signing a Non-Disclosure Agreement (NDA).



## 7-1 OVERVIEW

A FTDI device is composed of at least one interface. Each device interface is responsible for data transmission through a specific port. A FTDI device may have up to four different serial ports. Data transmission on a specific port is done through a pair of bulk IN/OUT endpoints. The device will use a *Vendor Specific* (0xFF) class code but the *Vendor ID* and the *Product ID* may be use instead to detect a FTDI device.

### 7-1-1 ENDPOINTS

A FTDI device is composed of the following endpoints:

- A pair of control IN and OUT endpoints called default endpoint.
- A pair of bulk IN and OUT endpoints by serial port (interface).

Table 7-1 indicates the different uses of the endpoints.

Endpoint	Direction	Used for
Control IN	Device to Host	Enumeration and FTDI class-specific requests
Control OUT	Host to Device	Enumeration and FTDI class-specific requests
Bulk IN	Device to Host	Receive serial status and data
Bulk OUT	Host to Device	Send data

Table 7-1 FTDI Endpoint Use

## 7-1-2 CLASS REQUESTS

There are numerous control requests used with FTDI chips. These requests and their descriptions are detailed in Table 7-2.

<b>Requests</b>	<b>Description</b>
Reset	Reset the communication port.
ModemCtrl	Set the modem control register.
SetFlowCtrl	Set flow control options.
SetBaudRate	Set the baud rate.
SetData	Set the data characteristics of the port.
GetModemStat	Retrieve the current value of the modem status register.
SetEventChar	Set the event character.
SetErrorChar	Set the error character.
SetLatTimer	Set the latency timer value.
GetLatTimer	Get the latency timer value.
SetBitMode	Set a special bit mode or turn on a special function.
GetBitMode	Return the current values on the data bus pins.
EEPROM readE2	Read the external EEPROM (if fitted), or the internal EEPROM of the FT232R.
EEPROM writeE2	Write to the external EEPROM (if fitted), or the internal EEPROM of the FT232R.
EEPROM eraseE2	Erase the external EEPROM (if fitted).

Table 7-2 **FTDI Class Requests**

### 7-1-3 SERIAL STATUS

The FTDI device generates a two-byte serial status every 16 milliseconds. Thus, the first two bytes of the data received on the bulk IN endpoint are reserved for this status.

The serial status is defined as shown in Table 7-3.

Offset	Modem Status Description (byte 0)	Line Status Description (byte 1)
B0	Full-speed, 64 bytes max packet size	Reserved
B1	High-speed, 512 bytes max packet size	Overrun Error (OE)
B2	Reserved	Parity Error (PE)
B3	Reserved	Framing Error (FE)
B4	Clear to Send (CTS)	Break Interrupt (BI)
B5	Data Set Ready (DSR)	Transmitter Holding Register (THRE)
B6	Ring Indicator (RI)	Transmitter Empty (TEMT)
B7	Data Carrier Detect (DCD)	Reserved

Table 7-3 FTDI Serial Status

## 7-2 CLASS IMPLEMENTATION

$\mu$ C/USB-Host Future Technology Devices International (FTDI) class is designed to support the basic vendor requests and multiple ports/interfaces FTDI Devices. This allows the user to specify the port on which data should be transmitted.

The FTDI class also handles the serial status generated by the device. The data received from the device will then be passed to the application in a raw format, letting the application parse it and handle it as desired. The class can also parse the status to detect line errors and notify the application.

## 7-3 CONFIGURATION AND INITIALIZATION

### 7-3-1 GENERAL CONFIGURATION

There are three configuration constants necessary to customize the FTDI host class. Those constants are located in the `usbh_cfg.h` file. Table 7-4 shows a description of the constants.

Constant	Description
<code>USBH_FTDI_CFG_MAX_DEV</code>	Configures the maximum number of FTDI functions the class can handle.
<code>USBH_FTDI_CFG_ID_VENDOR_CUSTOM</code>	Some FTDI device can use a custom vendor ID that can be set here. Default value is 0x0000.
<code>USBH_FTDI_CFG_ID_PRODUCT_CUSTOM</code>	Some FTDI device can use a custom product ID that can be set here. Default value is 0x0000.

Table 7-4 FTDI Configuration Constants

### 7-3-2 CLASS INITIALIZATION

In order to be integrated to the core and considered on a device connection, the FTDI class driver must be added to the core class driver list. This is done by calling `USBH_FTDI_Init()`. This function also registers the connection and disconnection callbacks to notify the application. This is described in Listing .

```

static USBH_FTDI_CALLBACKS App_USBH_FTDI_Callbacks = {
    App_USBH_FTDI_Conn,
    App_USBH_FTDI_Disconn
};
(1)

void App_USBH_FTDI_Init (USBH_ERR *p_err)
{
    ...

    USBH_FTDI_Init(&APP_USBH_FTDI_Callbacks,
        p_err);
(2)
}

```

```
static void App_USBH_FTDI_Conn (USBH_FTDI_HANDLE ftdi_handle)           (3)
{
    ...
}

static void App_USBH_FTDI_Disconn (USBH_FTDI_HANDLE ftdi_handle)       (4)
{
    ...
}
```

Listing 7-1 **FTDI Class Initialization**

- L7-1(1) Set the connection/disconnection callbacks.
- L7-1(2) Register the callbacks and the FTDI class driver.
- L7-1(3) Connection function callback.
- L7-1(4) Disconnection function callback.

## 7-4 DEMO APPLICATION

A simple demo application is provided with the host Future Technology Devices International (FTDI) class. This demo application can be used as a starting point to create your own application.

To properly setup the demo, you must connect the FTDI device to the board. Then, a serial null-modem cable must be connected between the device and the computer. On the computer, you may use *HyperTerminal* (or any similar serial COM port terminal emulator) to monitor the connection and send data to the demo application.

The demo application creates two tasks at initialization. A task for transmitting data periodically and one to receive data periodically. Upon connection of a FTDI device, the demo application will set up the modem connection as shown in Listing 7-3. After, the reception task will receive data periodically and display the status (if different from last status) and the received data in the output terminal. Listing 7-2 shows an example of output given by the demo application when two FTDI device are connected. The transmission task will send “*USB Host - FTDI Class Demo - Handle #x*” periodically where *x* is the FTDI function index.

```
----- SERIAL STATUS ----- (1)
Handle : 1
----- MODEM STATUS -----
Full-speed           : 1
High-speed           : 0
Clear to Send       : 1
Data Set Ready      : 1
Ring Indicator      : 0
Receive Line Signal Detect : 0
----- LINE STATUS -----
Overrun Error       : 0
Parity Error        : 0
Framing Error       : 0
Break Interrupt     : 0
Transmitter Holding Register : 1
Transmitter Empty   : 1
-----

----- SERIAL STATUS -----
Handle : 2
----- MODEM STATUS -----
Full-speed           : 1
High-speed           : 0
Clear to Send       : 1
Data Set Ready      : 1
Ring Indicator      : 0
Receive Line Signal Detect : 0
----- LINE STATUS -----
Overrun Error       : 0
Parity Error        : 0
Framing Error       : 0
Break Interrupt     : 0
Transmitter Holding Register : 1
Transmitter Empty   : 1
-----

Handle #1: Test: Receive data from device 1. (2)
Handle #2: Test: Receive data from device 2.
```

Listing 7-2 FTDI Demo Application Output

L7-2(1) Modem and line status. This is only displayed when the status changes.

L7-2(2) Received data. Display the handle number followed by the data received from the FTDI device.

```
static void App_USBH_FTDI_Conn (USBH_FTDI_HANDLE ftdi_handle) (1)
{
    USBH_ERR err;

    ...

    USBH_FTDI_Reset(ftdi_handle, (2)
                    USBH_FTDI_RESET_CTRL_SIO,
                    &err);
    if (err != USBH_ERR_NONE) { (3)
        APP_TRACE_INFO("Error in USBH_FTDI_Reset(): %d\r\n", err);
        return;
    }

    USBH_FTDI_ModemCtrlSet(ftdi_handle, (4)
                           (USBH_FTDI_CTRL_DTR_DISABLED | USBH_FTDI_MODEM_CTRL_RTS_DISABLED)
                           &err);
    if (err != USBH_ERR_NONE) {
        APP_TRACE_INFO("Error in USBH_FTDI_ModemCtrlSet(): %d\r\n", err);
        return;
    }

    USBH_FTDI_BaudRateSet(ftdi_handle, (5)
                         USBH_FTDI_BAUD_RATE_115200,
                         &err);
    if (err != USBH_ERR_NONE) {
        APP_TRACE_INFO("Error in USBH_FTDI_BaudRateSet(): %d\r\n", err);
        return;
    }

    USBH_FTDI_DataSet(ftdi_handle, (6)
                    8u,
                    USBH_FTDI_PARITY_NONE,
                    USBH_FTDI_STOP_BITS_1,
                    USBH_FTDI_DATA_BREAK_DISABLED,
                    &err);
    if (err != USBH_ERR_NONE) {
        APP_TRACE_INFO("Error in USBH_FTDI_DataSet(): %d\r\n", err);
        return;
    }
}
```

Listing 7-3 FTDI Demo Application

L7-3(1) This function is called upon device connection.

L7-3(2) Reset the RX and the TX buffer of the FTDI device.

- L7-3(3)      Display the error code on the terminal.
- L7-3(4)      Disable both hardware handshaking.
- L7-3(5)      Set the baud rate of the connection to 115 200 bits per second.
- L7-3(6)      Set the data characteristics. Here we use eight bits of data, no parity, one stop bit and no data break.

## 7-5 DEMO APPLICATION CONFIGURATION

Before running the FTDI demo application, you must configure it properly. The configuration constants that must be set are located in the `app_cfg.h` file. Table 7-5 lists the preprocessor constants that must be defined.

Preprocessor Constants	Description	Default Value
<code>APP_CFG_USBH_EN</code>	Enables $\mu$ C/USB-Host in your application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_FTDI_EN</code>	Enables FTDI demo application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_FTDI_RX_TASK_PRIO</code>	Reception task priority.	10
<code>APP_CFG_USBH_FTDI_TX_TASK_PRIO</code>	Transmission task priority.	11
<code>APP_CFG_USBH_FTDI_RXTX_TASK_STK_SIZE</code>	Reception and transmission task stack size.	1024

Table 7-5 Demo Application Configuration Constants





## Human Interface Device Class

This chapter describes the Human Interface Device (HID) class supported by  $\mu$ C/USB-Host. The HID implementation complies with the following specifications:

- *Device Class Definition for Human Interface Devices (HID), 6/27/01, Version 1.11.*
- *Universal Serial Bus HID Usage Tables, 10/28/2004, Version 1.12.*

The HID class encompasses devices used by humans to control computer operations. Keyboards, mice, pointing devices, game devices are some examples of typical HID devices. The HID class can also be used in a composite device that contains some controls such as knobs, switches, buttons and sliders. HID data can exchange data for any purpose using only control and interrupt transfers. The HID class is one of the oldest and most popular USB classes. This class also includes various types of output directed to the user information (e.g. LEDs on a keyboard).

## 8-1 OVERVIEW

A HID device is composed of the following endpoints:

- A pair of control IN and OUT endpoints called the default endpoint.
- An interrupt IN endpoint.
- An optional interrupt OUT endpoint.

Table 8-1 describes the use of the different endpoints:

Endpoint	Direction	Used for
Control IN	Device-to-host	Standard requests for enumeration, class-specific requests, and data communication (Input, Feature reports sent to the host with <code>GET_REPORT</code> request).
Control OUT	Host-to-device	Standard requests for enumeration, class-specific requests and data communication (Output, Feature reports received from the host with <code>SET_REPORT</code> request).
Interrupt IN	Device-to-host	Data communication (Input and Feature reports).
Interrupt OUT	Host-to-device	Data communication (Output and Feature reports).

Table 8-1 **HID Class Endpoints Use**

### 8-1-1 REPORT

A host and a HID device exchange data using reports. A report contains formatted data giving information about controls and other physical entities of the HID device. A control is manipulable by the user and operates an aspect of the device. For instance, a control can be a button on a mouse or a keyboard, a switch, etc. Other entities inform the user about the state of certain device's features. For instance, LEDs on a keyboard notify the user about the caps lock on, the numeric keypad active, etc.

The format and the use of a report data is understood by the host by analyzing the content of a *Report descriptor*. Analyzing the content is done by a parser. The Report descriptor describes the data provided by each control in a device. It is composed of *items*. An item is a piece of information about the device and consists of a 1-byte prefix and variable-length data. Refer to “*Device Class Definition for Human Interface Devices (HID) Version 1.11*”, section 5.6 and 6.2.2 for more details about the item format.

There are three principal types of items:

- *Main item* defines or groups certain types of data fields.
- *Global item* describes data characteristics of a control.
- *Local item* describes data characteristics of a control.

Each item type is defined by different functions. An item function can also be called a tag. An item function can be seen as a sub-item that belongs to one of the three principal item types. Table 8-2 gives a brief overview of the item’s functions in each item type. For a complete description of the items in each category, refer to *Device Class Definition for Human Interface Devices (HID) Version 1.11*, section 6.2.2.

Item type	Item function	Description
Main	Input	Describes information about the data provided by one or more physical controls.
	Output	Describes data sent to the device.
	Feature	Describes device configuration information sent to or received from the device which influences the overall behavior of the device or one of its components.
	Collection	Group related items (Input, Output or Feature).
	End of Collection	Closes a collection.

<b>Item type</b>	<b>Item function</b>	<b>Description</b>
Global	Usage Page	Identifies a function available within the device.
	Logical Minimum	Defines the lower limit of the reported values in logical units.
	Logical Maximum	Defines the upper limit of the reported values in logical units.
	Physical Minimum	Defines the lower limit of the reported values in physical units, that is the Logical Minimum expressed in physical units.
	Physical Maximum	Defines the upper limit of the reported values in physical units, that is the Logical Maximum expressed in physical units.
	Unit Exponent	Indicates the unit exponent in base 10. The exponent ranges from -8 to +7.
	Unit	Indicates the unit of the reported values. For instance, length, mass, temperature units, etc.
	Report Size	Indicates the size of the report fields in bits.
	Report ID	Indicates the prefix added to a particular report.
	Report Count	Indicates the number of data fields for an item.
	Push	Places a copy of the global item state table on the CPU stack.
	Pop	Replaces the item state table with the last structure from the stack.
Local	Usage	Represents an index to designate a specific Usage within a Usage Page. It indicates the vendor's suggested use for a specific control or group of controls. A usage supplies information to an application developer about what a control is actually measuring.
	Usage Minimum	Defines the starting usage associated with an array or bitmap.
	Usage Maximum	Defines the ending usage associated with an array or bitmap.
	Designator Index	Determines the body part used for a control. Index points to a designator in the Physical descriptor.
	Designator Minimum	Defines the index of the starting designator associated with an array or bitmap.
	Designator Maximum	Defines the index of the ending designator associated with an array or bitmap.
	String Index	String index for a String descriptor. It allows a string to be associated with a particular item or control.
	String Minimum	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap.
	String Maximum	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap.
	Delimiter	Defines the beginning or end of a set of local items.

Table 8-2 Item's Function Description for each Item Type

A control's data must define at least the following items:

- Input, Output or Feature Main items.
- Usage Local item.
- Usage Page Global item.
- Logical Minimum Global item.
- Logical Maximum Global item.
- Report Size Global item.
- Report Count Global item.

Figure 8-1 shows the representation of a Mouse Report descriptor content from a host HID parser perspective. The mouse has three buttons (left, right and wheel).

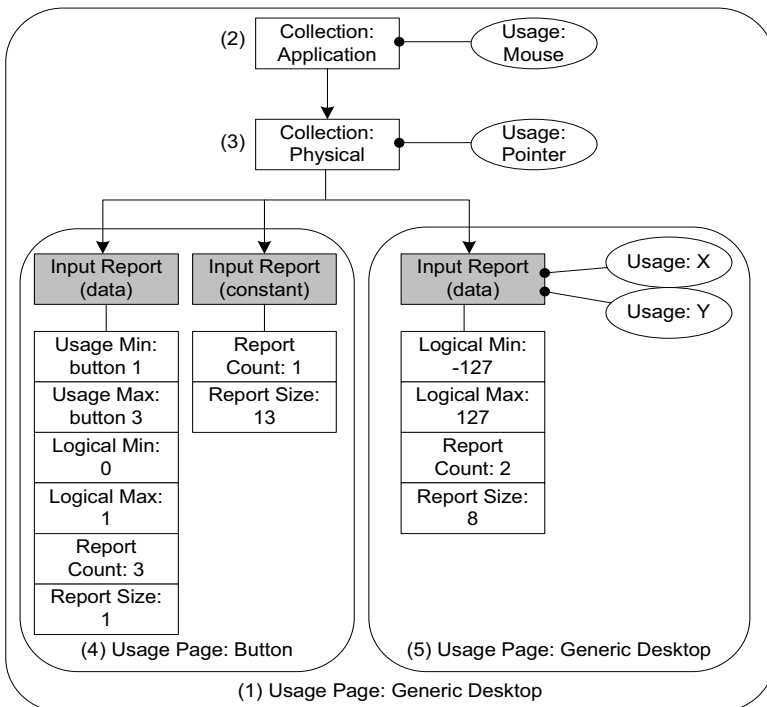


Figure 8-1 Report Descriptor Content from a Host HID Parser View

- F8-1(1) The *Usage Page* item function specifies the general function of the device. In this example, the HID device belongs to a generic desktop control.
- F8-1(2) The *Collection Application* groups Main items that have a common purpose and may be familiar to applications. In the diagram, the group is composed of three Input Main items. For this collection, the suggested use for the controls is a mouse as indicated by the *Usage* item.
- F8-1(3) Nested collections may be used to give more details about the use of a single control or group of controls to applications. In this example, the Collection Physical, nested into the Collection Application, is composed of the same 3 Input items forming the Collection Application. The *Collection Physical* is used for a set of data items that represent data points collected at one geometric point. In the example, the suggested use is a pointer as indicated by the Usage item. Here the pointer usage refers to the mouse position coordinates and the system software will translate the mouse coordinates in movement of the screen cursor.
- F8-1(4) Nested usage pages are also possible and give more details about a certain aspect within the general function of the device. In this case, two Input items are grouped and correspond to the buttons of the mouse. One Input item defines the three buttons of the mouse (right, left and wheel) in terms of number of data fields for the item (*Report Count* item), size of a data field (*Report Size* item) and possible values for each data field (*Usage Minimum* and *Maximum*, *Logical Minimum* and *Maximum* items). The other Input item is a 13-bit constant allowing the Input report data to be aligned on a byte boundary. This Input item is used only for padding purposes.
- F8-1(5) Another nested usage page referring to a generic desktop control is defined for the mouse position coordinates. For this usage page, the Input item describes the data fields corresponding to the x- and y-axis as specified by the two Usage items.

After analyzing the previous mouse Report descriptor content, the host's HID parser is able to interpret the Input report data sent by the device with an interrupt IN transfer or in response to a `GET_REPORT` request. The Input report data corresponding to the mouse Report descriptor shown in Figure 8-1 is presented in Table 8-3. The total size of the report data is 4 bytes. Different types of reports may be sent over the same endpoint. For the purpose of distinguishing the different types of reports, a 1-byte report ID prefix is added to the data report. If a report ID was used in the example of the mouse report, the total size of the report data would be 5 bytes.

Bit offset	Bit count	Description
0	1	Button 1 (left button).
1	1	Button 2 (right button).
2	1	Button 3 (wheel button).
3	13	Not used.
16	8	Position on axis X.
24	8	Position on axis Y.

Table 8-3 Input Report Sent to Host and Corresponding to the State of a 3-Buttons Mouse.

A Physical descriptor indicates the part or parts of the body intended to activate a control or controls. An application may use this information to assign a functionality to the control of a device. A Physical descriptor is an optional class-specific descriptor and most devices have little gain for using it. Refer to “Device Class Definition for Human Interface Devices (HID) Version 1.11” section 6.2.3 for more details about this descriptor.

## 8-2 CLASS IMPLEMENTATION

$\mu$ C/USB-Host Human Interface Device (HID) class is designed to support multiple interfaces/report IDs HID devices. It offers a report descriptor parser that will be able to determine the usage and the number of report IDs and types of the device. This allows the application to listen on specific input report IDs and send feature/output reports. The reports received from the device will be passed to your application in a raw format, letting the application parse it and handle any specific element(s).



## 8-3 CONFIGURATION AND INITIALIZATION

### 8-3-1 GENERAL CONFIGURATION

There are many configuration constants necessary to customize the HID host class. These constants are located in the `usbh_cfg.h` file. Table 8-2 shows their description.

Constant	Description
<code>USBH_HID_CFG_MAX_DEV</code>	Maximum number of HID functions the class can handle.
<code>USBH_HID_CFG_MAX_NBR_APP_COLL</code>	Maximum number of application collections per HID function.
<code>USBH_HID_CFG_MAX_NBR_REPORT_ID</code>	Maximum number of report IDs per HID function.
<code>USBH_HID_CFG_MAX_NBR_REPORT_FMT</code>	Maximum number of report formats per application collection.
<code>USBH_HID_CFG_MAX_NBR_USAGE</code>	Maximum number of usages per report format.
<code>USBH_HID_CFG_MAX_TX_BUF_SIZE</code>	Length in octets of transmission buffer (used with output and feature reports).
<code>USBH_HID_CFG_MAX_RX_BUF_SIZE</code>	Length in octets of reception buffer (must be at least equal to the length of the longest input report).
<code>USBH_HID_CFG_MAX_NBR_RXCB</code>	Maximum number of application callback functions associated to input report per HID function.
<code>USBH_HID_CFG_MAX_REPORT_DESC_LEN</code>	Maximum length in octets of the report descriptor.
<code>USBH_HID_CFG_MAX_ERR_CNT</code>	Maximum number of communication errors that can occur before the communication is stopped.
<code>USBH_HID_CFG_MAX_GLOBAL</code>	Maximum number of push/pop items.
<code>USBH_HID_CFG_MAX_COLL</code>	Maximum number of collections.

Figure 8-2 HID Configuration Constants

---

## 8-3-2 CLASS INITIALIZATION

In order to be integrated to the core and considered on a device connection, the HID class driver must be added to the core class driver list. This is done by calling `USBH_ClassDrvReg()` and is described in Listing 8-1.

```
USBH_ERR App_USBH_HID_Init (void)
{
    USBH_ERR    err;

    ...

    err = USBH_ClassDrvReg(    &USBH_HID_ClassDrv,           (1)
                             App_USBH_HID_ClassNotify,      (2)
                             (void *)0);                   (3)

    return(err);
}
```

Listing 8-1 **HID Initialization**

- L8-1(1) First parameter is the HID class driver structure. It is defined in `usbh_hid.h`.
- L8-1(2) Second parameter is a pointer to the application's callback function. This function will be called upon HID device connection/disconnection.
- L8-1(3) Last parameter is an optional pointer to application specific data.

### 8-3-3 DEVICE CONNECTION AND DISCONNECTION HANDLING

Upon connection/disconnection of a HID device, your application will be notified via a callback function. Listing describes the operations that must be performed at that moment.

```

static void App_USBH_HID_ClassNotify (void      *p_class_dev,
                                       CPU_INT08U is_conn,
                                       void      *p_ctx)
{
    CPU_INT08U      nbr_report_id;
    CPU_INT08U      report_id_cnt;
    USBH_HID_REPORT_ID *p_report_id_array;
    USBH_HID_REPORT_ID *p_report_id;
    USBH_HID_DEV     *p_hid_dev;
    USBH_ERR         err;

    (void)&p_ctx;
    p_hid_dev = (USBH_HID_DEV *)p_class_dev;
    switch (is_conn) {
        case USBH_CLASS_DEV_STATE_CONN:
            err = USBH_HID_RefAdd(p_hid_dev);           (1)
            if (err != USBH_ERR_NONE) {
                /* $$$ Handle error */
                return;
            }

            err = USBH_HID_IdleSet(p_hid_dev, 0u, 0u);   (2)
            if (err == USBH_ERR_EP_STALL) {
                /* $$$ Handle error. */                 (3)
            } else if (err != USBH_ERR_NONE) {
                /* $$$ Handle error. */
                return;
            }

            err = USBH_HID_Init(p_hid_dev);             (4)
            if (err != USBH_ERR_NONE) {
                /* $$$ Handle error */
                return;
            }
    }
}

```

```

err = USBH_HID_GetReportIDArray(p_hid_dev,                (5)
                                &p_report_id_array,
                                &nbr_report_id);

if (err != USBH_ERR_NONE) {
    /* $$$ Handle error */
    return;
}

for (report_id_cnt = 0u; report_id_cnt < nbr_report_id; report_id_cnt++) {
    p_report_id = &p_report_id_array[report_id_cnt];
    if (p_report_id->Type != 0x08u) {                (6)
        continue;
    }

    if (p_hid_dev->Usage == USBH_HID_APP_USAGE_KBD) {
        err = USBH_HID_RegRxCB(                        (7)
                                p_hid_dev,
                                p_report_id->ReportID,
                                (USBH_HID_RXCB_FNCT)App_USBH_KBD_CallBack,
                                (void *)p_hid_dev);
    } else if (p_hid_dev->Usage == USBH_HID_APP_USAGE_MOUSE) {
        err = USBH_HID_RegRxCB(
                                p_hid_dev,
                                p_report_id->ReportID,
                                (USBH_HID_RXCB_FNCT)App_USBH_MouseCallBack,
                                (void *)p_hid_dev);
    }
}
break;

case USBH_CLASS_DEV_STATE_DISCONN:
    USBH_HID_RefRel(p_hid_dev);                (8)
    break;
}
}

```

Listing 8-2 HID Device Connection/Disconnection

L8-2(1) First step is to add an application reference to this HID device by calling the function `USBH_HID_RefAdd()`.

L8-2(2) You should then immediately call the function `USBH_HID_IdleSet()`. This will set the idle time of the HID device. Some devices may fail to respond correctly if the *SetIdle* request is not issued at that moment. For more information of the *SetIdle* request, see *Device Class Definition for Human Interface Devices (HID), Version 1.11, section 7.2.4*.

- L8-2(3) Some HID devices do not support the *SetIdle* request and thus will reply with a *STALL* handshake. This is normal and the function should not return in this case.
- L8-2(4) The function `USBH_HID_Init()` should then be called. At this moment, the report descriptor will be requested and parsed.
- L8-2(5) In order to listen to incoming input reports sent by the device, your application needs to set callback function(s). In order to retrieve the reports list, the function `USBH_HID_GetReportIDArray()` should be called.
- L8-2(6) Skip report IDs that are not of *INPUT* type.
- L8-2(7) You can set different application callback functions for each report ID depending on the device usage. The usage defined in `p_hid_dev->Usage` is the first usage local item of the report descriptor. Your application callback function can be set by calling the function `USBH_HID_RegRxCB()`. Note that this callback function will receive the rough report and will have to parse it manually.
- L8-2(8) Release the application's reference to the HID device.

## 8-4 DEMO APPLICATION

A simple demo application is provided with the host Human Interface Device (HID) class. This demo application can be used as a starting point to create your own application. This demo application expect a certain report format and is compatible with the majority of the mice and keyboards using US layout available on the market. However, there might be mice or keyboards that use slightly different reports and thus you will have to parse the report differently. Also note that you will have to write your own report parser for other device types like gamepads.

Depending on the device type (mouse or keyboard), the demo application will output status on the default output console. If a mouse is connected, the output will look like Listing 8-3. If the device is a keyboard, the console will display the key pressed.

```

Pointer at (x, y) = (-202, 308)
Pointer at (x, y) = (-221, 307)
Pointer at (x, y) = (-225, 306)
Pointer at (x, y) = (-228, 305)
Pointer at (x, y) = (-230, 304)
Pointer at (x, y) = (-231, 303)
Pointer at (x, y) = (-232, 302)
Left  button pressed
Left  button released
Pointer at (x, y) = (-243, 296)
Pointer at (x, y) = (-247, 295)
Pointer at (x, y) = (-253, 294)
Pointer at (x, y) = (-256, 293)
Right button pressed
Pointer at (x, y) = (-258, 292)
Pointer at (x, y) = (-260, 291)
Pointer at (x, y) = (-261, 290)
Right button released

```

Listing 8-3 Demo Application Output when a Mouse is Connected

### 8-4-1 DEMO APPLICATION CONFIGURATION

Before running the HID demo application, you must configure it properly. The configuration constants that must be set are located in the `app_cfg.h` file. Table 8-4 lists the preprocessor constants that must be defined.

Preprocessor Constants	Description	Default Value
APP_CFG_USBH_EN	Enables $\mu$ C/USB-Host in your application.	DEF_ENABLED
APP_CFG_USBH_HID_EN	Enables HID demo application.	DEF_ENABLED

Table 8-4 Demo Application Configuration Constants

## Mass Storage Class

This section describes the Mass Storage Class (MSC) supported by  $\mu$ C/USB-Host. The MSC implementation offered by  $\mu$ C/USB-Host is in compliance with the following specifications:

- *Universal Serial Bus Mass Storage Class Specification Overview*, Revision 1.3 Sept. 5, 2008.
- *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, Revision 1.0 Sept. 31, 1999.

MSC is a protocol that enables the transfer of information between a USB device and a host. The information is anything that can be stored electronically: executable programs, source code, documents, images, configuration data, or other text or numeric data. The USB device appears as an external storage medium to the host.

A file system defines how the files are organized in the storage media. The USB mass storage class specification does not require any particular file system to be used on conforming devices. Instead, it provides a simple interface to read and write sectors of data using the Small Computer System Interface (SCSI) transparent command set.

The USB mass storage host class supports two transport protocols:

- Bulk-Only Transport (BOT)
- Control/Bulk/Interrupt (CBI) Transport.

The mass storage device class supported by  $\mu$ C/USB-Host implements the SCSI transparent command set using the BOT protocol only, which signifies that only bulk endpoints will be used to transmit data and status information. The MSC implementation supports only one logical unit.

## 9-1 OVERVIEW

### 9-1-1 MASS STORAGE CLASS PROTOCOL

The MSC protocol is composed of three phases:

- The Command Transport
- The Data Transport
- The Status Transport

Mass storage commands are sent by the host through a structure called the Command Block Wrapper (CBW). For commands requiring a data transport stage, the host will attempt to send or receive the exact number of bytes from the device as specified by the length and flag fields of the CBW. After the data transport stage, the host attempts to receive a Command Status Wrapper (CSW) from the device detailing the status of the command as well as any data residue (if any). For commands that do not include a data transport stage, the host attempts to receive the CSW directly after CBW is sent. The protocol is detailed in Figure 9-1.

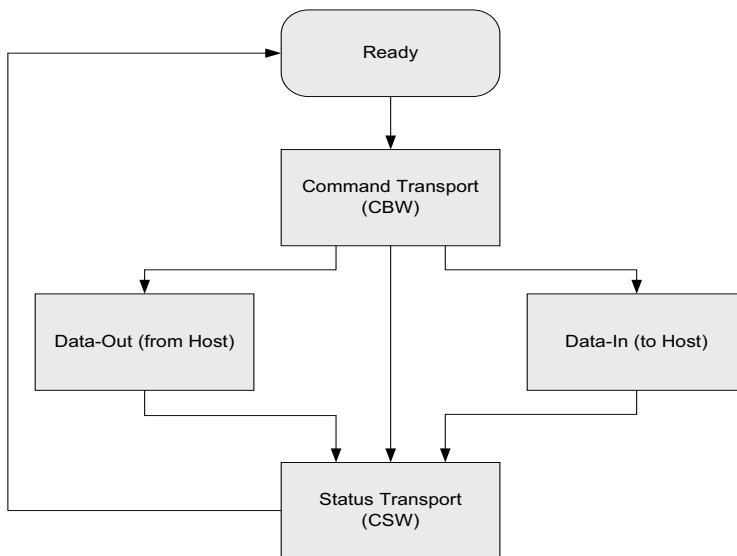


Figure 9-1 MSC Protocol



## 9-1-2 ENDPOINTS

On the host side, in compliance with the BOT specification, a MSC device is composed of the following endpoints:

- A pair of control IN and OUT endpoints called default endpoint.
- A pair of bulk IN and OUT endpoints.

Table 9-1 indicates the different uses of the endpoints.

Endpoint	Direction	Used for
Control IN Control OUT	Device to Host Host to Device	Enumeration and MSC class-specific requests
Bulk IN Bulk OUT	Device to Host Host to Device	Receive CSW and data Send CBW and data

Table 9-1 **MSC Endpoint Use**

## 9-1-3 MASS STORAGE CLASS REQUESTS

There are two defined control requests for the MSC BOT protocol. These requests and their descriptions are detailed in Table 9-2.

Class Requests	Description
Bulk-Only Mass Storage Reset	This request is used to reset the mass storage device and its associated interface. This request readies the device to receive the next command block.
Get Max LUN	This request is used to return the highest logical unit number (LUN) supported by the device. For example, a device with LUN 0 and LUN 1 will return a value of 1. A device with a single logical unit will return 0 or stall the request. The maximum value that can be returned is 15.

Table 9-2 **Mass Storage Class Requests**

### **9-1-4 SMALL COMPUTER SYSTEM INTERFACE (SCSI)**

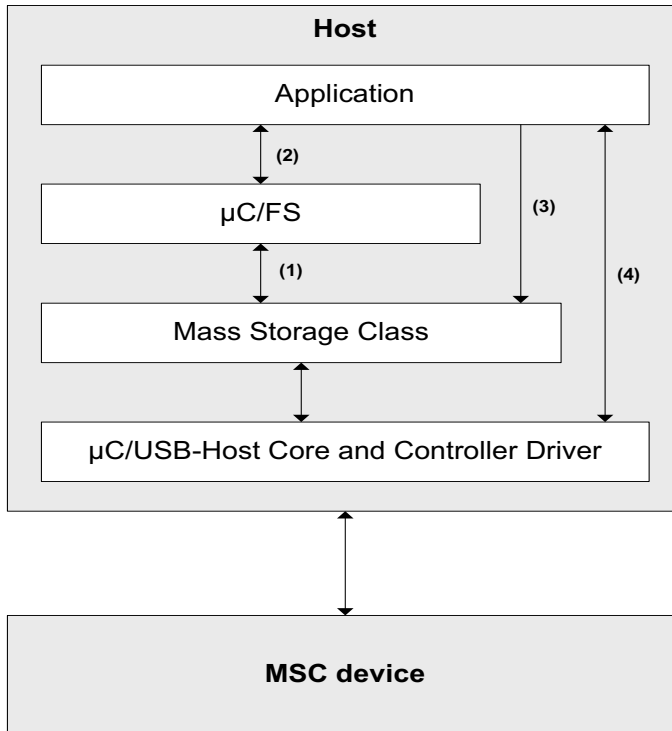
SCSI is a set of standards for handling communication between computers and peripheral devices. These standards include commands, protocols, electrical interfaces and optical interfaces. Storage devices that use other hardware interfaces such as USB, use SCSI commands for obtaining device/host information and controlling the device's operation and transferring blocks of data in the storage media.

SCSI commands cover a vast range of device types and functions and as such, devices need a subset of these commands. In general, the following commands are necessary for basic communication:

- INQUIRY
- READ CAPACITY(10)
- READ(10)
- REQUEST SENSE
- TEST UNIT READY
- WRITE(10)

### **9-2 CLASS IMPLEMENTATION**

$\mu$ C/USB-Host Mass Storage Class (MSC) requires a File System (FS) to work properly. Micrium's  $\mu$ C/FS supports  $\mu$ C/USB-Host MSC and is the recommended FS. Figure 9-2 describes how your application interacts with MSC, the core module, and  $\mu$ C/FS.

Figure 9-2 **MSC Interactions**

- F9-2(1) μC/FS sees the USB MSC class as a storage device driver.
- F9-2(2) All the operations on files and folders from your application are done via μC/FS. Your application cannot access a MSC device using μC/USB-Host MSC without the use of a file system. For more information on how to perform file and folder operations, refer to the μC/FS user manual.
- F9-2(3) Upon device connection, your application must add a reference on the device and release it on device disconnection.
- F9-2(4) At initialization, your application must register the MSC driver to the core.

## 9-3 CONFIGURATION AND INITIALIZATION

### 9-3-1 GENERAL CONFIGURATION

There is only one configuration constant necessary to customize the MSC host class. This constant is located in the `usbh_cfg.h` file. Table 9-3 shows a description of this constant.

Constant	Description
<code>USBH_MSC_CFG_MAX_DEV</code>	Configures the maximum number of MSC functions the class can handle.

Figure 9-3 **MSC Configuration Constant**

### 9-3-2 CLASS INITIALIZATION

In order to be integrated to the core and considered on a device connection, the MSC class driver must be added to the core class driver list. This is done by calling `USBH_ClassDrvReg()` and is described in Listing 9-1.

```

USBH_ERR App_USBH_MSC_Init (void)
{
    USBH_ERR    err;

    ...

    err = USBH_ClassDrvReg(    &USBH_MSC_ClassDrv,          (1)
                              App_USBH_MSC_ClassNotify,      (2)
                              (void *)0);                   (3)

    return (err);
}

```

Listing 9-1 **MSC Initialization**

- L9-1(1) First parameter is MSC class driver structure. It is defined in `usbh_msc.h`.
- L9-1(2) Second parameter is a pointer to the application's callback function. This function will be called upon MSC device connection/disconnection.
- L9-1(3) Last parameter is an optional pointer to application specific data.

### 9-3-3 DEVICE CONNECTION AND DISCONNECTION HANDLING

Upon connection/disconnection of a MSC device, your application will be notified via a callback function. Listing 9-2 describes the operations that must be performed at that moment.

```

static void App_USBH_MSC_ClassNotify (void      *p_class_dev,
                                       CPU_INT08U is_conn,
                                       void      *p_ctx)
{
    USBH_MSC_DEV *p_msc_dev;
    USBH_ERR      err_usbh;
    FS_ERR        err_fs;
    CPU_INT32U    unit_nbr;

    (void)&p_ctx;
    p_msc_dev = (USBH_MSC_DEV *)p_class_dev;

    switch (is_conn) {
        case USBH_CLASS_DEV_STATE_CONN:
            err_usbh = USBH_MSC_RefAdd(p_msc_dev);           (1)
            if (err_usbh != USBH_ERR_NONE) {
                /* $$$ Handle error */
                return;
            }

            unit_nbr = FSDev_MSC_DevOpen(p_msc_dev, &err_fs); (2)
            if (err_fs != FS_ERR_NONE) {
                FSDev_MSC_DevClose(p_msc_dev);
                USBH_MSC_RefRel(p_msc_dev);
                /* $$$ Handle error */
                return;
            }
            break;

        case USBH_CLASS_DEV_STATE_DISCONN:
            FSDev_MSC_DevClose(p_msc_dev);                 (3)
            USBH_MSC_RefRel(p_msc_dev);
            break;

        default:
            break;
    }
}

```

Listing 9-2 MSC Device Connection/Disconnection

- L9-2(1) On a MSC device connection, the first step is to add an application reference to this MSC device by calling the function `USBH_MSC_RefAdd( )`.
- L9-2(2) Second step consists of allocating/opening a MSC device in the File System (FS). This is done by calling `FSDev_MSC_DevOpen( )`. If this call is successful, you will now be able to communicate with the device using  $\mu$ C/FS. The connected MSC device will have the following mounting point: *msc:<unit\_nbr>://*.
- L9-2(3) On MSC device disconnection, your application must de-allocate/close the MSC device in the FS and release its reference.

## 9-4 DEMO APPLICATION

A simple demo application is provided with the host Mass Storage Class (MSC). This demo application can be used as a starting point to create your own application.

Upon connection of a MSC device, the demo application will open a file, perform a write/read operation, and close the file. Listing 9-3 describes the file operations that are performed by the demo application.

```

#define USBH_MSC_DEMO_FS_EXAMPLE_FILE                "msc:x:\\MSPrint.txt"
#define APP_USBH_MSC_FS_BUF_SIZE                    49u

static CPU_INT08U App_USBH_MSC_BufTx[] = "This is USB Mass Storage Demo sample output file.";

static void App_USBH_MSC_FileTask (void *p_ctx)
{
    ...

    (void)&p_ctx;
    (void)Str_Copy(&name[0u], APP_USBH_MSC_FS_EXAMPLE_FILE);

    while (DEF_TRUE) {
        unit_nbr = (CPU_INT32U)USBH_OS_MsgQueueGet(App_USBH_MSC_DevQ,           (1)
                                                    0u,
                                                    &err_usbh);

        if (err_usbh != USBH_ERR_NONE) {
            continue;
        }

        name[4u] = ASCII_CHAR_DIGIT_ZERO + (CPU_CHAR)unit_nbr;
        p_file = FSFile_Open(name,                                           (2)
                              (FS_FILE_ACCESS_MODE_CREATE |
                               FS_FILE_ACCESS_MODE_WR      |
                               FS_FILE_ACCESS_MODE_RD),
                              &err_fs);

        if (err_fs != FS_ERR_NONE) {
            continue;
        }

        len_wr = FSFile_Wr(          p_file,                                (3)
                                (void *)&App_USBH_MSC_BufTx[0u],
                                APP_USBH_MSC_FS_BUF_SIZE,
                                &err_fs);

        if ((err_fs != FS_ERR_NONE      ) ||
            (len_wr != APP_USBH_MSC_FS_BUF_SIZE)) {
            FSFile_Close(p_file, &err_fs);
            continue;
        }

        FSFile_PosSet(p_file,
                     0u,
                     FS_FILE_ORIGIN_START,
                     &err_fs);

        if (err_fs != FS_ERR_NONE) {
            FSFile_Close(p_file, &err_fs);
            continue;
        }
    }
}

```

```

len_rd = FSFile_Rd(          p_file,                (4)
                      (void *)&App_USBH_MSC_BufRx[0u],
                      APP_USBH_MSC_FS_BUF_SIZE,
                      &err_fs);
if ((err_fs != FS_ERR_NONE      ) ||
    (len_rd != APP_USBH_MSC_FS_BUF_SIZE)) {
    FSFile_Close(p_file, &err_fs);
    continue;
}

APP_TRACE_INFO(("Comparing original data and data read from USB drive... "));
cmp = Mem_Cmp((void *)&App_USBH_MSC_BufTx[0u],      (5)
             (void *)&App_USBH_MSC_BufRx[0u],
             APP_USBH_MSC_FS_BUF_SIZE);
if (cmp == DEF_YES) {
    APP_TRACE_INFO(("Passed\n\r"));
} else {
    APP_TRACE_INFO(("Failed!\n\r"));
}

FSFile_Close(p_file, &err_fs);
}
}

```

Listing 9-3 MSC Demo Application

- L9-3(1) The application's file task pends on a queue. Once a device is connected, the queue is posted from the application class notification callback with the unit number of the connected MSC device.
- L9-3(2) The file named **MSPrint.txt** is opened in the root folder of the MSC device. If the file is not present, it will be created.
- L9-3(3) The string *This is the USB Mass Storage Demo sample output file.* is written to the **MSPrint.txt** file.
- L9-3(4) The content of the file is read back.
- L9-3(5) The written and read content of the file are compared. If they match, the demo application indicates a success.



---

## 9-4-1 DEMO APPLICATION CONFIGURATION

Before running the MSC demo application, you must configure it properly. The configuration constants that must be set are located in the `app_cfg.h` file. Table 9-3 lists the preprocessor constants that must be defined.

Preprocessor Constants	Description	Default Value
<code>APP_CFG_USBH_EN</code>	Enables $\mu$ C/USB-Host in your application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_MSC_EN</code>	Enables MSC demo application.	<code>DEF_ENABLED</code>
<code>APP_CFG_USBH_MSC_FILE_TASK_Prio</code>	MSC application task priority.	20
<code>APP_CFG_USBH_MSC_FILE_TASK_STK_SIZE</code>	MSC application task stack size.	1000

Table 9-3 Demo Application Configuration Constants

In order to use the demo application, you must properly configure  $\mu$ C/FS. Refer to the  $\mu$ C/FS user manual for more information on how to configure, initialize and use this product.



# Chapter 10

## Porting $\mu$ C/USB-Host to your Kernel

$\mu$ C/USB-Host requires a Kernel (also known as a Real-Time Operating System, RTOS). In order to make it usable with nearly any kernels available on the market, it has been designed to be easily portable. Micrium provides a port for both  $\mu$ C/OS-II and  $\mu$ C/OS-III and recommends using one of these kernels. In case you need to use another kernel, this chapter will explain you how to port  $\mu$ C/USB-Host to your kernel.

### 10-1 OVERVIEW

In order to be portable to other kernels,  $\mu$ C/USB-Host uses an abstraction layer to instantiate and use kernel services. Figure 10-1 shows how the different modules of  $\mu$ C/USB-Host interacts with the kernel via the abstraction layer.

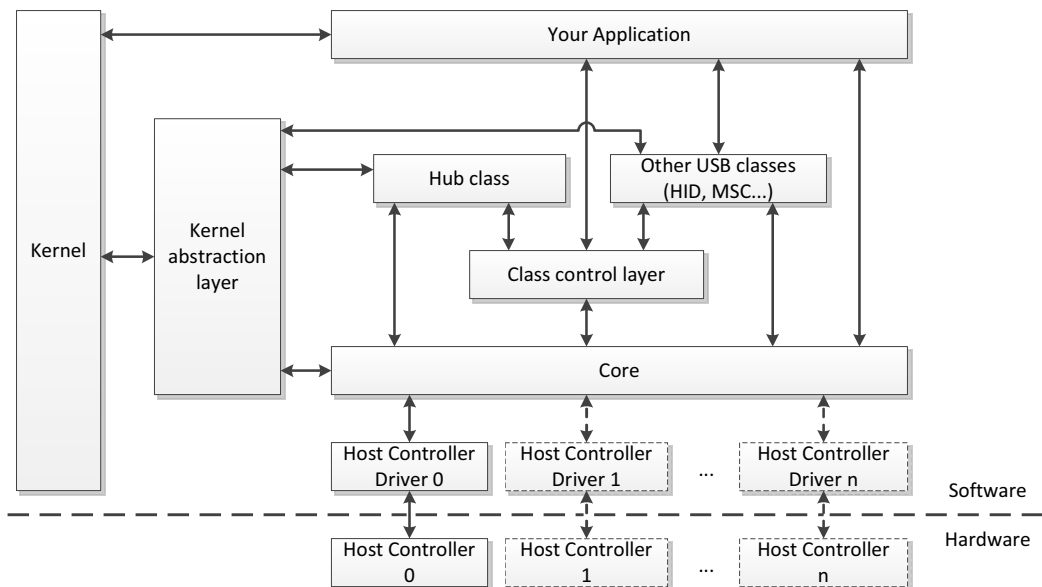


Figure 10-1  $\mu$ C/USB-Host Interactions with the Kernel

Each USB class implementation and the core layer interacts with the kernel abstraction layer. The kernel abstraction layer is shared by all the modules. It offers a unified API to the stack to access common kernel services such as task creation and semaphore creation/pend/post, etc. Your application should not use the abstraction layer. It should interact directly with the kernel. However, for maintenance simplicity, the demo applications provided by Micrium uses the kernel abstraction layer.

$\mu$ C/USB-Host requires a kernel that minimally offers the following services:

- Task creation
- Semaphores
- Mutex

Plus, the Mass Storage Class (MSC) demo application provided by Micrium requires message queuing.

## 10-2 PORTING THE STACK TO YOUR KERNEL

A template kernel abstraction layer file is provided with  $\mu$ C/USB-Host. It is located in the following folder:

```
\Micrium\Software\uC-USB-Host-V3\OS\Template
```

### 10-2-1 TASK CREATION

$\mu$ C/USB-Host requires a few tasks to work properly:

- One for the *async* task
- One for the *hub* task
- One for the Communication Device Class (CDC) demo application (optional)
- One for the Mass Storage Class (MSC) demo application (optional)

To instantiate these tasks,  $\mu$ C/USB-Host will call the `USBH_OS_TaskCreate()` function twice. This function must be implemented in your kernel abstraction layer.

For further details on how to implement this function, refer to Appendix A, “Kernel Abstraction Functions” on page 160.

## 10-2-2 SEMAPHORE

µC/USB-Host requires several semaphores to work properly:

- One for the *async* task
- One for the *hub* task
- One per endpoint
- One for the CDC demo application (optional)

Listing 10-1 shows how the number of semaphores required should be computed in your kernel abstraction layer (if necessary).

```
#define USBH_OS_SEM_REQUIRED (3u + (((USBH_CFG_MAX_NBR_EPS * USBH_CFG_MAX_NBR_IFS) + 1u) * \
                                USBH_CFG_MAX_NBR_DEVS))
```

Listing 10-1 **Semaphore Number Computation**

Table 10-1 summarizes the semaphores related functions that must be implemented in your kernel abstraction layer.

Function	Description
USBH_OS_SemCreate()	Creates/instantiates a semaphore.
USBH_OS_SemDestroy()	Destroys a semaphore.
USBH_OS_SemWait()	Waits/pends on a semaphore.
USBH_OS_SemWaitAbort()	Aborts pend on a semaphore.
USBH_OS_SemPost()	Posts a semaphore.

Table 10-1 **Semaphore API**

For further details on how to implement these functions, refer to Appendix A, “Kernel Abstraction Functions” on page 160.

### 10-2-3 MUTEX

µC/USB-Host requires several mutexes (mutual exclusion) to work properly:

- One per Host Controller Driver (HCD)
- One per device
- One per USB function
- One per endpoint

Listing 10-2 shows how the number of mutex required should be computed in your kernel abstraction layer (if necessary).

```
#define USBH_OS_MUTEX_REQUIRED (((USBH_CFG_MAX_NBR_EPS * USBH_CFG_MAX_NBR_IFS) + 1u) * \
                                USBH_CFG_MAX_NBR_DEVS) + \
                                USBH_CFG_MAX_NBR_DEVS + USBH_CFG_MAX_NBR_HC + \
                                USBH_CDC_CFG_MAX_DEV + USBH_HID_CFG_MAX_DEV + \
                                USBH_MSC_CFG_MAX_DEV)
```

Listing 10-2 **Mutex Number Computation**

Table 10-2 summarizes the mutex related functions that must be implemented in your kernel abstraction layer.

Function	Description
USBH_OS_MutexCreate()	Creates/instantiates a mutex.
USBH_OS_MutexLock()	Locks a mutex.
USBH_OS_MutexUnlock()	Unlocks a mutex.
USBH_OS_MutexDestroy()	Destroys a mutex.

Table 10-2 **Mutex API**

For further details on how to implement these functions, refer to Appendix A, “Kernel Abstraction Functions” on page 160.

---

## 10-2-4 MESSAGE QUEUE

µC/USB-Host's MSC demo application requires a message queue to work properly.

Table 10-3 summarizes the message queue related functions that must be implemented in your kernel abstraction layer. If the MSC demo application is not used within your project, you can leave these functions empty.

Function	Description
USBH_OS_MsgQueueCreate()	Creates/instantiates a message queue.
USBH_OS_MsgQueuePut()	Posts a message to the queue.
USBH_OS_MsgQueueGet()	Pends on the queue and returns the first element.

Table 10-3 **Message Queues API**

For further details on how to implement these functions, refer to Appendix A, “Kernel Abstraction Functions” on page 160.





## Appendix

# A

## Core API Reference

This appendix provides a reference to the  $\mu$ C/USB-Host core layer API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

## A-1 HOST FUNCTIONS

### A-1-1 USBH\_Init()

Allocates and initializes resources required by the USB Host stack.

#### FILES

usbh\_core.h/usbh\_core.c

#### PROTOTYPE

```
USBH_ERR USBH_Init (USBH_KERNEL_TASK_INFO *async_task_info,  
                   USBH_KERNEL_TASK_INFO *hub_task_info);
```

#### ARGUMENTS

**async\_task\_info**            Structure that contains information on asynchronous task.

**hub\_task\_info**             Structure that contains information on hub task.

#### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_ALLOC  
USBH_ERR_INVALID_ARG  
USBH_ERR_CLASS_DRV_ALLOC  
USBH_ERR_OS_SIGNAL_CREATE  
USBH_ERR_OS_TASK_CREATE
```

#### CALLERS

Application.

---

## **NOTES / WARNINGS**

`USBH_Init()` must be called:

- Only once from a product's application.
- After product's OS has been initialized.

## **A-1-2 USBH\_VersionGet()**

Get the  $\mu$ C/USB-Host software version.

### **FILES**

usbh\_core.h/usbh\_core.c

### **PROTOTYPE**

```
CPU_INT32U USBH_VersionGet (void)
```

### **ARGUMENTS**

None.

### **RETURNED VALUE**

$\mu$ C/USB-Host version.

### **CALLERS**

Application.

### **NOTES / WARNINGS**

The value returned is multiplied by 10000. For example, version 3.40.02, would be returned as 34002.

---

### **A-1-3 USBH\_Suspend()**

Suspends the USB Host stack by calling suspend for every class driver loaded followed by a call to the suspend function of each host controller.

#### **FILES**

usbh\_core.h/usbh\_core.c

#### **PROTOTYPE**

```
USBH_ERR USBH_Suspend (void)
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

Error code from this function.

`USBH_ERR_NONE`

Host Controller Driver error code.

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

This call will move all the connected devices to the *suspend* state as well.

### **A-1-4 USBH\_Resume()**

Resumes the USB Host stack by calling every host controller resume function and then calling resume for every class driver loaded.

#### **FILES**

usbh\_core.h/usbh\_core.c

#### **PROTOTYPE**

```
USBH_ERR USBH_Resume (void)
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

Error code from this function.

`USBH_ERR_NONE`

Host Controller Driver error code.

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

Calling this function will also resume the bus activity with all the devices connected.

---

## A-2 HOST CONTROLLER FUNCTIONS

### A-2-1 USBH\_HC\_Add()

Adds a host controller.

#### FILES

usbh\_core.h/usbh\_core.c

#### PROTOTYPE

```
CPU_INT08U USBH_HC_Add (USBH_HC_CFG      *p_hc_cfg,  
                        USBH_HC_DRV_API  *p_drv_api,  
                        USBH_HC_RH_API   *p_hc_rh_api,  
                        USBH_HC_BSP_API  *p_hc_bsp_api,  
                        USBH_ERR        *p_err);
```

#### ARGUMENTS

<code>p_hc_cfg</code>	Pointer to specific USB host controller configuration.
<code>p_drv_api</code>	Pointer to specific USB host controller driver API.
<code>p_hc_rh_api</code>	Pointer to specific USB host controller root hub driver API.
<code>p_hc_bsp_api</code>	Pointer to specific USB host controller board-specific API.
<code>p_err</code>	Pointer to variable that will receive the return error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_HC_ALLOC  
USBH_ERR_DEV_ALLOC  
USBH_ERR_OS_SIGNAL_CREATE  
Host Controller Driver error code
```

**RETURNED VALUE**

Host Controller index      if host controller successfully added.

USBH\_HC\_NBR\_NONE      Otherwise.

**CALLERS**

Application.

**NOTES / WARNINGS**

None.



---

## A-2-2 USBH\_HC\_Start()

Starts the given host controller.

### FILES

usbh\_core.h/usbh\_core.c

### PROTOTYPE

```
USBH_ERR USBH_HC_Start (CPU_INT08U hc_nbr);
```

### ARGUMENTS

hc\_nbr        Host controller number.

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DESC_INVALID  
USBH_ERR_CFG_MAX_CFG_LEN  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
USBH_ERR_DRIVER_NOT_FOUND  
USBH_ERR_OS_SIGNAL_CREATE  
Host Controller Driver error code
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

### **A-2-3 USBH\_HC\_Stop()**

Stops the given host controller.

#### **FILES**

usbh\_core.h/usbh\_core.c

#### **PROTOTYPE**

```
USBH_ERR USBH_HC_Stop (CPU_INT08U hc_nbr);
```

#### **ARGUMENTS**

hc\_nbr        Host controller number.

#### **RETURNED VALUE**

Error code from this function:

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

Host Controller Driver error code

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

None.

---

## A-2-4 USBH\_HC\_PortEn()

Enable specified port of given host controller's root hub.

### FILES

usbh\_core.h/usbh\_core.c

### PROTOTYPE

```
USBH_ERR USBH_HC_PortEn (CPU_INT08U hc_nbr,  
                        CPU_INT08U port_nbr);
```

### ARGUMENTS

**hc\_nbr** Host controller number.

**port\_nbr** Port number.

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host Controller Driver error code
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

## A-2-5 USBH\_HC\_PortDis()

Disable specified port of given host controller's root hub.

### FILES

usbh\_core.h/usbh\_core.c

### PROTOTYPE

```
USBH_ERR USBH_HC_PortDis (CPU_INT08U hc_nbr,  
                          CPU_INT08U port_nbr);
```

### ARGUMENTS

**hc\_nbr** Host controller number.

**port\_nbr** Port number.

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host Controller Driver error code
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

---

## A-2-6 USBH\_HC\_FrameNbrGet()

Gets current frame number.

### FILES

usbh\_core.h/usbh\_core.c

### PROTOTYPE

```
CPU_INT32U USBH_HC_FrameNbrGet (CPU_INT08U hc_nbr,  
                                USBH_ERR *p_err);
```

### ARGUMENTS

**hc\_nbr** Host controller number.

**p\_err** Pointer to variable that will receive the return error code from this function :

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

Host Controller Driver error code

### RETURNED VALUE

Curent frame number processed by the Host Controller.

### CALLERS

Application.

### NOTES / WARNINGS

None.

## A-3 CLASS MANAGEMENT FUNCTIONS

### A-3-1 USBH\_ClassDrvReg()

Registers a class driver to the USB host stack.

#### FILES

usbh\_class.h/usbh\_class.c

#### PROTOTYPE

```
USBH_ERR USBH_ClassDrvReg (USBH_CLASS_DRV      *p_class_drv,  
                           USBH_CLASS_NOTIFY_FNCT class_notify_funct,  
                           void                  *p_class_notify_ctx);
```

#### ARGUMENTS

**p\_class\_drv**                    Pointer to the class driver.

**class\_notify\_funct**            Class notification function pointer.

**p\_class\_notify\_ctx**            Class notification function context pointer.

#### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_CLASS_DRV_ALLOC
```

#### CALLERS

Application.

#### NOTES / WARNINGS

None.

---

## A-3-2 USBH\_ClassDrvUnreg()

Unregisters a class driver from the USB host stack.

### FILES

usbh\_class.h/usbh\_class.c

### PROTOTYPE

```
USBH_ERR USBH_ClassDrvUnreg (USBH_CLASS_DRV *p_class_drv)
```

### ARGUMENTS

`p_class_drv`                      Pointer to the class driver.

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_CLASS_DRV_NOT_FOUND
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

## **A-4 KERNEL ABSTRACTION FUNCTIONS**

### **A-4-1 USBH\_OS\_LayerInit()**

Initializes the kernel abstraction layer.

#### **FILES**

usbh\_os.h/usbh\_os.c

#### **PROTOTYPE**

```
USBH_ERR USBH_OS_LayerInit (void)
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_ALLOC
```

#### **CALLERS**

USBH\_Init().

#### **NOTES / WARNINGS**

None.



---

## A-4-2 USBH\_OS\_VirToBus()

Converts from virtual address to physical address if the operating system uses virtual memory.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
void *USBH_OS_VirToBus (void *x)
```

### ARGUMENTS

**x** Virtual address to convert

### RETURNED VALUE

The corresponding physical address

### CALLERS

- OHCI driver
- EHCI driver

### NOTES / WARNINGS

Most of the embedded operating systems don't use virtual memory addressing. Hence, this function can be implemented as shown in Listing A-1.

```
void *USBH_OS_VirToBus (void *x)
{
    return (x);
}
```

Listing A-1 **Typical USBH\_OS\_VirToBus() Function Implementation**

### A-4-3 USBH\_OS\_BusToVir()

Converts from physical address to virtual address if the operating system uses virtual memory.

#### FILES

usbh\_os.h/usbh\_os.c

#### PROTOTYPE

```
void *USBH_OS_BusToVir (void *x)
```

#### ARGUMENTS

**x** Physical address to convert

#### RETURNED VALUE

The corresponding virtual address

#### CALLERS

- OHCI driver
- EHCI driver

#### NOTES / WARNINGS

Most of the embedded kernels don't use virtual memory addressing. Hence, this function can be implemented as shown in Listing A-2.

```
void *USBH_OS_BusToVir (void *x)
{
    return (x);
}
```

Listing A-2 **Typical** USBH\_OS\_BusToVir() **Function Implementation**

---

## A-4-4 USBH\_OS\_TaskCreate()

Creates a task.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
USBH_ERR USBH_OS_TaskCreate (CPU_CHAR      *p_name,  
                             CPU_INT32U    prio,  
                             USBH_TASK_FNCT task_funct,  
                             void          *p_data,  
                             CPU_INT32U    *p_stk,  
                             CPU_INT32U    stk_size,  
                             USBH_HTASK    *p_task)
```

### ARGUMENTS

- p\_name**        Pointer to a string that contains a name to assign to the task (can be NULL)
- prio**         Priority of the task
- task\_funct**   Pointer to the task's function body
- p\_data**       Pointer to the data that is passed to the task function
- p\_stk**        Pointer to the beginning of the task's stack
- stk\_size**     Size of the task's stack
- p\_task**       Pointer to a variable that will receive the handle of the task

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_ALLOC  
USBH_ERR_OS_TASK_CREATE
```

## **CALLERS**

- Core layer
- Micrium's CDC and MSC demo applications

## **NOTES / WARNINGS**

None.

---

## **A-4-5 USBH\_OS\_DlyMS()**

Delays the current task by the specified delay in milliseconds.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
void USBH_OS_DlyMS (CPU_INT32U dly)
```

### **ARGUMENTS**

dly            Delay, in milliseconds

### **RETURNED VALUE**

None.

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

## **A-4-6 USBH\_OS\_DlyUS()**

Delays the current task by the specified delay in microseconds.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
void USBH_OS_DlyUS (CPU_INT32U dly)
```

### **ARGUMENTS**

dly            Delay, in microseconds

### **RETURNED VALUE**

None.

### **CALLERS**

Host Controller Driver.

### **NOTES / WARNINGS**

None.

---

## **A-4-7 USBH\_OS\_MutexCreate()**

Creates a mutex.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_MutexCreate (USBH_HMUTEX *p_mutex)
```

### **ARGUMENTS**

**p\_mutex**      Pointer to variable that will receive handle of the mutex.

### **RETURNED VALUE**

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_ALLOC  
USBH_ERR_OS_SIGNAL_CREATE
```

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

## **A-4-8 USBH\_OS\_MutexLock()**

Acquires/locks a mutex.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_MutexLock (USBH_HMUTEX mutex)
```

### **ARGUMENTS**

**mutex**           Mutex handle.

### **RETURNED VALUE**

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_OS_TIMEOUT  
USBH_ERR_OS_FAIL
```

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.



---

## **A-4-9 USBH\_OS\_MutexUnlock()**

Releases a mutex.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_MutexUnlock (USBH_HMUTEX mutex)
```

### **ARGUMENTS**

**mutex**            Mutex handle.

### **RETURNED VALUE**

Error code from this function:

USBH\_ERR\_NONE

USBH\_ERR\_OS\_FAIL

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

## **A-4-10 USBH\_OS\_MutexDestroy()**

Destroys a mutex.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_MutexDestroy (USBH_HMUTEX mutex)
```

### **ARGUMENTS**

**mutex**            Mutex handle.

### **RETURNED VALUE**

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_OS_FAIL  
USBH_ERR_FREE
```

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

---

## A-4-11 USBH\_OS\_SemCreate()

Creates a semaphore initialized with the given count.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
USBH_ERR USBH_OS_SemCreate (USBH_HSEM *p_sem,  
                             CPU_INT32U cnt)
```

### ARGUMENTS

**p\_sem** Pointer to variable that will receive handle of the semaphore

**cnt** Count value with which the semaphore will be initialized

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_OS_SIGNAL_CREATE  
USBH_ERR_ALLOC
```

### CALLERS

Various.

### NOTES / WARNINGS

None.

## **A-4-12 USBH\_OS\_SemDestroy()**

Destroys a semaphore.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_SemDestroy (USBH_HSEM sem)
```

### **ARGUMENTS**

sem            Semaphore handle

### **RETURNED VALUE**

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_OS_FAIL  
USBH_ERR_FREE
```

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

---

## A-4-13 USBH\_OS\_SemWait()

Pends on a semaphore.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
USBH_ERR USBH_OS_SemWait (USBH_HSEM sem,  
                          CPU_INT32U timeout)
```

### ARGUMENTS

**sem** Semaphore handle

**timeout** Timeout value, expressed in milliseconds

### RETURNED VALUE

Error code from this function:

```
USBH_ERR_NONE  
USBH_ERR_OS_TIMEOUT  
USBH_ERR_OS_ABORT  
USBH_ERR_OS_FAIL
```

### CALLERS

Various.

### NOTES / WARNINGS

None.

## **A-4-14 USBH\_OS\_SemWaitAbort()**

Aborts a semaphore and resumes all tasks pending on it.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_SemWaitAbort (USBH_HSEM sem)
```

### **ARGUMENTS**

**sem** Semaphore handle

### **RETURNED VALUE**

Error code from this function:

USBH\_ERR\_NONE

USBH\_ERR\_OS\_FAIL

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

---

## **A-4-15 USBH\_OS\_SemPost()**

Posts a semaphore.

### **FILES**

usbh\_os.h/usbh\_os.c

### **PROTOTYPE**

```
USBH_ERR USBH_OS_SemPost (USBH_HSEM sem)
```

### **ARGUMENTS**

`sem` Semaphore handle

### **RETURNED VALUE**

Error code from this function:

`USBH_ERR_NONE`

`USBH_ERR_OS_FAIL`

### **CALLERS**

Various.

### **NOTES / WARNINGS**

None.

## A-4-16 USBH\_OS\_MsgQueueCreate()

Creates a message queue.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
USBH_HQUEUE USBH_OS_MsgQueueCreate (void      **p_start,  
                                     CPU_INT16U size,  
                                     USBH_ERR  *p_err)
```

### ARGUMENTS

**p\_start**      Pointer to the base address of the message queue storage area

**size**          Size of the queue

**p\_err**          Pointer to variable that will receive the return error code from this function

USBH\_ERR\_NONE  
USBH\_ERR\_ALLOC  
USBH\_ERR\_OS\_SIGNAL\_CREATE

### RETURNED VALUE

Handle of the message queue.

### CALLERS

Micrium's MSC demo application.

### NOTES / WARNINGS

This function can be left empty if Micrium's MSC demo application is not used.



---

## A-4-17 USBH\_OS\_MsgQueuePut()

Posts a message to a message queue.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
USBH_ERR USBH_OS_MsgQueuePut (USBH_HQUEUE msg_q,  
                               void        *p_msg)
```

### ARGUMENTS

**msg\_q**            Message queue handle

**p\_msg**            Pointer to the message to post

### RETURNED VALUE

Error code from this function:

USBH\_ERR\_NONE

USBH\_ERR\_OS\_FAIL

### CALLERS

Micrium's MSC demo application.

### NOTES / WARNINGS

This function can be left empty if Micrium's MSC demo application is not used.

## A-4-18 USBH\_OS\_MsgQueueGet()

Pends on a queue and retrieves the first message from it.

### FILES

usbh\_os.h/usbh\_os.c

### PROTOTYPE

```
void *USBH_OS_MsgQueueGet (USBH_HQUEUE msg_q,  
                           CPU_INT32U timeout,  
                           USBH_ERR *p_err)
```

### ARGUMENTS

**msg\_q** Message queue handle

**timeout** Pointer to the message to post

**p\_err** Pointer to variable that will receive the return error code from this function

USBH\_ERR\_NONE  
USBH\_ERR\_OS\_TIMEOUT  
USBH\_ERR\_OS\_ABORT  
USBH\_ERR\_OS\_FAIL

### RETURNED VALUE

Pointer to the first message from the queue.

### CALLERS

Micrium's MSC demo application.

### NOTES / WARNINGS

This function can be left empty if Micrium's MSC demo application is not used.

## Appendix

# B

## CDC API Reference

This appendix provides a reference to the  $\mu$ C/USB-Host Communication Device Class (CDC) and the Abstract Control Model (ACM) subclass API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

## **B-1 CDC FUNCTIONS**

### **B-1-1 USBH\_CDC\_RefAdd()**

Increments the CDC device's application reference counter.

#### **FILES**

usbh\_cdc.h/usbh\_cdc.c

#### **PROTOTYPE**

```
USBH_ERR USBH_CDC_RefAdd (USBH_CDC_DEV *p_cdc_dev)
```

#### **ARGUMENTS**

`p_cdc_dev`                      Pointer to the CDC device.

#### **RETURNED VALUE**

Error code from this function.

`USBH_ERR_NONE`

`USBH_ERR_INVALID_ARG`

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

None.

---

## **B-1-2 USBH\_CDC\_RefRel()**

Decrements the CDC device's application reference counter. Frees the device if there is not anymore references to it.

### **FILES**

usbh\_cdc.h/usbh\_cdc.c

### **PROTOTYPE**

```
USBH_ERR USBH_CDC_RefRel (USBH_CDC_DEV *p_cdc_dev)
```

### **ARGUMENTS**

`p_cdc_dev`                      Pointer to the CDC device.

### **RETURNED VALUE**

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_FREE
```

### **CALLERS**

Application.

### **NOTES / WARNINGS**

None.

### **B-1-3 USBH\_CDC\_SubclassGet()**

Retrieves CDC device's subclass code.

#### **FILES**

usbh\_cdc.h/usbh\_cdc.c

#### **PROTOTYPE**

```
USBH_ERR USBH_CDC_SubclassGet(USBH_CDC_DEV *p_cdc_dev,  
                               CPU_INT08U *p_subclass)
```

#### **ARGUMENTS**

**p\_cdc\_dev**                      Pointer to the CDC device.

**p\_subclass**                    Pointer to variable that will receive the subclass code.

#### **RETURNED VALUE**

Error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

None.

---

## **B-1-4 USBH\_CDC\_ProtocolGet()**

Retrieves CDC device's protocol code.

### **FILES**

usbh\_cdc.h/usbh\_cdc.c

### **PROTOTYPE**

```
USBH_ERR USBH_CDC_ProtocolGet(USBH_CDC_DEV *p_cdc_dev,  
                               CPU_INT08U *p_protocol)
```

### **ARGUMENTS**

**p\_cdc\_dev**                      Pointer to the CDC device.

**p\_protocol**                    Pointer to variable that will receive the protocol code.

### **RETURNED VALUE**

Error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

### **CALLERS**

Application.

### **NOTES / WARNINGS**

None.

## **B-2 ACM FUNCTIONS**

### **B-2-1 USBH\_CDC\_ACM\_GlobalInit()**

Initializes all the USB CDC ACM structures and global variables.

#### **FILES**

usbh\_acm.h/usbh\_acm.c

#### **PROTOTYPE**

```
USBH_ERR USBH_CDC_ACM_GlobalInit (void)
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_ALLOC
```

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

This function must be called only once at initialization.



---

## **B-2-2 USBH\_CDC\_ACM\_Add()**

Allocates memory for a CDC ACM device structure, reads ACM descriptor from the interface and retrieves ACM events and requests supported by the device.

### **FILES**

usbh\_acm.h/usbh\_acm.c

### **PROTOTYPE**

```
USBH_CDC_ACM_DEV *USBH_CDC_ACM_Add (USBH_CDC_DEV *p_cdc_dev,  
                                     USBH_ERR *p_err)
```

### **ARGUMENTS**

**p\_cdc\_dev**                      Pointer to the CDC device.

**p\_err**                              Pointer to variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_ALLOC  
USBH\_ERR\_DESC\_INVALID  
USBH\_ERR\_INVALID\_ARG

### **RETURNED VALUE**

Pointer to the CDC ACM device.

### **CALLERS**

Application.

### **NOTES / WARNINGS**

None.

### **B-2-3 USBH\_CDC\_ACM\_Remove()**

Frees CDC ACM device.

#### **FILES**

usbh\_acm.h/usbh\_acm.c

#### **PROTOTYPE**

```
USBH_ERR USBH_CDC_ACM_Remove (USBH_CDC_ACM_DEV *p_cdc_acm_dev)
```

#### **ARGUMENTS**

`p_cdc_acm_dev`                      Pointer to the CDC ACM device.

#### **RETURNED VALUE**

Error code from this function.

`USBH_ERR_NONE`

`USBH_ERR_FREE`

`USBH_ERR_INVALID_ARG`

#### **CALLERS**

Application.

#### **NOTES / WARNINGS**

None.



## B-2-5 USBH\_CDC\_ACM\_LineCodingSet()

Specifies typical asynchronous line-character formatting properties by sending a *SetLineCoding* class request to the device.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_LineCodingSet (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                     CPU_INT32U      baud_rate,  
                                     CPU_INT08U      stop_bits,  
                                     CPU_INT08U      parity_val,  
                                     CPU_INT08U      data_bits)
```

### ARGUMENTS

**p\_cdc\_acm\_dev**                      Pointer to the CDC ACM device.

**baud\_rate**                          Baud rate in bits per second.

```
USBH_CDC_ACM_LINE_CODING_BAUDRATE_110  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_300  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_1200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_2400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_4800  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_9600  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_19200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_38400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_56700  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_115200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_230400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_460800  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_921600
```

---

`stop_bits`

Stop bits value.

`USBH_CDC_ACM_LINE_CODING_STOP_BIT_1`  
`USBH_CDC_ACM_LINE_CODING_STOP_BIT_1_5`  
`USBH_CDC_ACM_LINE_CODING_STOP_BIT_2`

`parity_val`

Parity value.

`USBH_CDC_ACM_LINE_CODING_PARITY_NONE`  
`USBH_CDC_ACM_LINE_CODING_PARITY_ODD`  
`USBH_CDC_ACM_LINE_CODING_PARITY_EVEN`  
`USBH_CDC_ACM_LINE_CODING_PARITY_MARK`  
`USBH_CDC_ACM_LINE_CODING_PARITY_SPACE`

`data_bits`

Number of data bits.

`USBH_CDC_ACM_LINE_CODING_DATA_BITS_5`  
`USBH_CDC_ACM_LINE_CODING_DATA_BITS_6`  
`USBH_CDC_ACM_LINE_CODING_DATA_BITS_7`  
`USBH_CDC_ACM_LINE_CODING_DATA_BITS_8`

## RETURNED VALUE

Error code from this function.

`USBH_ERR_NONE`  
`USBH_ERR_NOT_SUPPORTED`  
`USBH_ERR_INVALID_ARG`  
`USBH_ERR_DEV_NOT_READY`  
`USBH_ERR_UNKNOWN`  
`USBH_ERR_EP_INVALID_STATE`  
`USBH_ERR_HC_IO`  
`USBH_ERR_EP_STALL`

Host Controller Driver error code

## CALLERS

Application.

**NOTES / WARNINGS**

The *SetLineCoding* request is described in *Universal Serial Bus Communications Class Subclass Specification for PSTN Devices, revision 1.2 section 6.3.10*.

---

## B-2-6 USBH\_CDC\_ACM\_LineCodingGet()

Retrieves current device's asynchronous line-character formatting properties by sending a *GetLineCoding* class request to the device.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_LineCodingGet (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                     CPU_INT32U      *p_baud_rate,  
                                     CPU_INT08U      *p_stop_bits,  
                                     CPU_INT08U      *p_parity_val,  
                                     CPU_INT08U      *p_data_bits)
```

### ARGUMENTS

**p\_cdc\_acm\_dev** Pointer to the CDC ACM device.

**p\_baud\_rate** Pointer to variable that will receive the current baud rate in bits per second.

```
USBH_CDC_ACM_LINE_CODING_BAUDRATE_110  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_300  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_1200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_2400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_4800  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_9600  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_19200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_38400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_56700  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_115200  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_230400  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_460800  
USBH_CDC_ACM_LINE_CODING_BAUDRATE_921600
```

`p_stop_bits` Pointer to variable that will receive the current stop bits value.

```
USBH_CDC_ACM_LINE_CODING_STOP_BIT_1
USBH_CDC_ACM_LINE_CODING_STOP_BIT_1_5
USBH_CDC_ACM_LINE_CODING_STOP_BIT_2
```

`p_parity_val` Pointer to variable that will receive the current parity value.

```
USBH_CDC_ACM_LINE_CODING_PARITY_NONE
USBH_CDC_ACM_LINE_CODING_PARITY_ODD
USBH_CDC_ACM_LINE_CODING_PARITY_EVEN
USBH_CDC_ACM_LINE_CODING_PARITY_MARK
USBH_CDC_ACM_LINE_CODING_PARITY_SPACE
```

`p_data_bits` Pointer to variable that will receive the current number of data bits.

```
USBH_CDC_ACM_LINE_CODING_DATA_BITS_5
USBH_CDC_ACM_LINE_CODING_DATA_BITS_6
USBH_CDC_ACM_LINE_CODING_DATA_BITS_7
USBH_CDC_ACM_LINE_CODING_DATA_BITS_8
```

## RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE
USBH_ERR_NOT_SUPPORTED
USBH_ERR_INVALID_ARG
USBH_ERR_DEV_NOT_READY
USBH_ERR_UNKNOWN
USBH_ERR_EP_INVALID_STATE
USBH_ERR_HC_IO
USBH_ERR_EP_STALL
```

Host Controller Driver error code

## CALLERS

Application.



---

**NOTES / WARNINGS**

The *GetLineCoding* request is described in *Universal Serial Bus Communications Class Subclass Specification for PSTN Devices, revision 1.2, section 6.3.11*.

## B-2-7 USBH\_CDC\_ACM\_LineStateSet()

Generates RS-232/V.24 style control signals by sending a *SetControlLineState* request to the device.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_LineStateSet (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                     CPU_INT08U      dtr_bit,  
                                     CPU_INT08U      rts_bit)
```

### ARGUMENTS

- |                            |  |
|----------------------------|--|
| <code>p_cdc_acm_dev</code> | Pointer to the CDC ACM device.   |
| <code>dtr_bit</code>       | Indicates to DCE if DTE is present or not. This signal corresponds to V.24 signal 108/2 and RS-232 signal DTR.<br><br>USBH_CDC_ACM_DTR_SET<br>USBH_CDC_ACM_DTR_CLR |
| <code>rts_bit</code>       | Carrier control for half duplex modems. This signal corresponds to V.24 signal 105 and RS-232 signal RTS.<br><br>USBH_CDC_ACM_RTS_SET<br>USBH_CDC_ACM_RTS_CLR      |

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_NOT_SUPPORTED  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN
```

---

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_HC\_IO

USBH\_ERR\_EP\_STALL

Host Controller Driver error code

## **CALLERS**

Application.

## **NOTES / WARNINGS**

The *SetControlLineState* request is described in *Universal Serial Bus Communications Class Subclass Specification for PSTN Devices, revision 1.2, section 6.3.12*.

## **B-2-8 USBH\_CDC\_ACM\_BreakSend()**

Sends special carrier modulation that generates an RS-232 style break.

### **FILES**

usbh\_acm.h/usbh\_acm.c

### **PROTOTYPE**

```
USBH_ERR USBH_CDC_ACM_BreakSend (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                  CPU_INT16U break_time)
```

### **ARGUMENTS**

**p\_cdc\_acm\_dev**                      Pointer to the CDC ACM device.

**break\_time**                        Duration of the break signal, in milliseconds.

### **RETURNED VALUE**

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_NOT_SUPPORTED  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host Controller Driver error code
```

### **CALLERS**

Application.

### **NOTES / WARNINGS**

The *SendBreak* request is described in *Universal Serial Bus Communications Class Subclass Specification for PSTN Devices, revision 1.2, section 6.3.13*.

---

## B-2-9 USBH\_CDC\_ACM\_CmdSend()

Sends an encapsulated command to the device using a class-specific request.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_CmdSend (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                CPU_INT08U      *p_buf,  
                                CPU_INT32U      buf_len)
```

### ARGUMENTS

**p\_cdc\_acm\_dev**            Pointer to the CDC ACM device.  
**p\_buf**                    Pointer to buffer that contains the command.  
**buf\_len**                 Buffer length in octets.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host Controller Driver error code
```

### CALLERS

Application.

### NOTES / WARNINGS

The *SendEncapsulatedCommand* request is described in *Universal Serial Bus Class Definitions for Communications Devices, revision 1.2, section 6.2.1*.

## **B-2-10 USBH\_CDC\_ACM\_RespRx()**

Receives encapsulated response from device.

### **FILES**

usbh\_acm.h/usbh\_acm.c

### **PROTOTYPE**

```
USBH_ERR USBH_CDC_ACM_RespRx (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                               CPU_INT08U *p_buf,  
                               CPU_INT32U buf_len)
```

### **ARGUMENTS**

<code>p_cdc_acm_dev</code>	Pointer to the CDC ACM device.
<code>p_buf</code>	Pointer to buffer that will receive the response from the device.
<code>buf_len</code>	Buffer length in octets.

### **RETURNED VALUE**

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host Controller Driver error code
```

### **CALLERS**

Application.

### **NOTES / WARNINGS**

The *GetEncapsulatedResponse* request is described in *Universal Serial Bus Class Definitions for Communications Devices, revision 1.2, section 6.2.2*.

---

## B-2-11 USBH\_CDC\_ACM\_DataTx()

Sends data to CDC ACM device. This function is blocking.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
CPU_INT32U  USBH_CDC_ACM_DataTx (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                void *p_buf,  
                                CPU_INT32U  buf_len,  
                                USBH_ERR    *p_err)
```

### ARGUMENTS

<code>p_cdc_acm_dev</code>	Pointer to the CDC ACM device.
<code>p_buf</code>	Pointer to transmit buffer.
<code>buf_len</code>	Buffer length in octets.
<code>p_err</code>	Variable that will receive the return error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_EP_INVALID_TYPE  
USBH_ERR_EP_INVALID_STATE  
Host Controller Driver error code
```

### RETURNED VALUE

Number of octets transmitted.

### CALLERS

Application.

### NOTES / WARNINGS

None.

## **B-2-12 USBH\_CDC\_ACM\_DataRx()**

Receives data from CDC ACM device. This function is blocking.

### **FILES**

usbh\_acm.h/usbh\_acm.c

### **PROTOTYPE**

```
CPU_INT32U  USBH_CDC_ACM_DataRx (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
                                void *p_buf,  
                                CPU_INT32U  buf_len,  
                                USBH_ERR    *p_err)
```

### **ARGUMENTS**

<code>p_cdc_acm_dev</code>	Pointer to the CDC ACM device.
<code>p_buf</code>	Pointer to buffer that will contain the received data.
<code>buf_len</code>	Buffer length in octets.
<code>p_err</code>	Variable that will receive the return error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_EP_INVALID_TYPE  
USBH_ERR_EP_INVALID_STATE  
Host Controller Driver error code
```

### **RETURNED VALUE**

Number of octets received.

### **CALLERS**

Application.

### **NOTES / WARNINGS**

None.



---

## B-2-13 USBH\_CDC\_ACM\_DataTxAsync()

Sends data to CDC ACM device. This function is non-blocking.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_DataTxAsync (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
void *p_buf,  
CPU_INT32U buf_len,  
USBH_CDC_DATA_NOTIFY tx_cmpl_notify,  
void *p_tx_cmpl_arg)
```

### ARGUMENTS

<code>p_cdc_acm_dev</code>	Pointer to the CDC ACM device.
<code>p_buf</code>	Pointer to buffer that contains data to transmit.
<code>buf_len</code>	Buffer length in octets.
<code>tx_cmpl_notify</code>	Function that will be invoked upon completion of transmit operation.
<code>p_tx_cmpl_arg</code>	Pointer to application's argument that will be passed as parameter of <code>tx_cmpl_notify</code> .

## **RETURNED VALUE**

Error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_EP\_INVALID\_TYPE

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_ALLOC

USBH\_ERR\_UNKNOWN

Host Controller Driver error code

## **CALLERS**

Application.

## **NOTES / WARNINGS**

None.

---

## B-2-14 USBH\_CDC\_ACM\_DataRxAsync()

Receives data from CDC ACM device. This function is non-blocking.

### FILES

usbh\_acm.h/usbh\_acm.c

### PROTOTYPE

```
USBH_ERR USBH_CDC_ACM_DataRxAsync (USBH_CDC_ACM_DEV *p_cdc_acm_dev,  
void *p_buf,  
CPU_INT32U buf_len,  
USBH_CDC_DATA_NOTIFY rx_cmpl_notify,  
void *p_rx_cmpl_arg)
```

### ARGUMENTS

<code>p_cdc_acm_dev</code>	Pointer to the CDC ACM device.
<code>p_buf</code>	Pointer to buffer that will contain the received data.
<code>buf_len</code>	Buffer length in octets.
<code>rx_cmpl_notify</code>	Function that will be invoked upon completion of receive operation.
<code>p_rx_cmpl_arg</code>	Pointer to application's argument that will be passed as parameter of <code>rx_cmpl_notify</code> .

## **RETURNED VALUE**

Error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_EP\_INVALID\_TYPE

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_ALLOC

USBH\_ERR\_UNKNOWN

Host Controller Driver error code

## **CALLERS**

Application.

## **NOTES / WARNINGS**

None.

# C

## FTDI API Reference

This appendix provides a reference to the  $\mu$ C/USB-Host Future Technology Devices International (FTDI) class API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

## C-1 FTDI FUNCTIONS

### C-1-1 USBH\_FTDI\_Init()

Initializes all the USB FTDI structures and global variables. This function also registers the FTDI class driver to the USB Host core.

#### FILES

usbh\_ftdi.h/usbh\_ftdi.c

#### PROTOTYPE

```
void USBH_FTDI_Init (USBH_FTDI_CALLBACKS *p_ftdi_callbacks,  
                    USBH_ERR *p_err)
```

#### ARGUMENTS

**p\_ftdi\_callbacks** Pointer to the FTDI callback functions structure to notify the application when a device is connected/disconnected.

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_ALLOC  
USBH\_ERR\_DEV\_ALLOC  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_CLASS\_DRV\_ALLOC  
USBH\_ERR\_OS\_SIGNAL\_CREATE

#### RETURNED VALUE

None.

#### CALLERS

Application.

---

**NOTES / WARNINGS**

This function must be called only once at initialization.

## C-1-2 USBH\_FTDI\_Reset()

Resets the selected communication port.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_Reset (USBH_FTDI_HANDLE ftdi_handle,  
                    CPU_INT16U reset_ctrl,  
                    USBH_ERR *p_err)
```

### ARGUMENTS

**ftdi\_handle** Handle on FTDI device.

**reset\_ctrl** Select buffer to reset.

USBH\_FTDI\_RESET\_CTRL\_SIO  
USBH\_FTDI\_RESET\_CTRL\_RX  
USBH\_FTDI\_RESET\_CTRL\_TX

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY  
Host controller driver error code

### RETURNED VALUE

None.

### CALLERS

Application.



---

## **NOTES / WARNINGS**

The *Reset* request is described in *API for FTxxxx Devices Application Note AN\_115, revision 1.1, section 3.1*.

### C-1-3 USBH\_FTDI\_ModemCtrlSet()

Sets the modem control register of the communication port.

#### FILES

usbh\_ftdi.h/usbh\_ftdi.c

#### PROTOTYPE

```
void USBH_FTDI_ModemCtrlSet (USBH_FTDI_HANDLE  ftdi_handle,
                             CPU_INT16U        modem_ctrl,
                             USBH_ERR          *p_err)
```

#### ARGUMENTS

**ftdi\_handle**                    Handle on FTDI device.

**modem\_ctrl**                    Modem control code. This is a bitmap of the following possible values:

```
USBH_FTDI_MODEM_CTRL_DTR_SET
USBH_FTDI_MODEM_CTRL_DTR_RESET
USBH_FTDI_MODEM_CTRL_DTR_ENABLED
USBH_FTDI_MODEM_CTRL_DTR_DISABLED
USBH_FTDI_MODEM_CTRL_RTS_SET
USBH_FTDI_MODEM_CTRL_RTS_RESET
USBH_FTDI_MODEM_CTRL_RTS_ENABLED
USBH_FTDI_MODEM_CTRL_RTS_DISABLED
```

**p\_err**                         Pointer to the variable that will receive the return error code from this function.

```
USBH_ERR_NONE
USBH_ERR_INVALID_ARG
USBH_ERR_DEV_NOT_READY
Host controller driver error code
```

---

## **RETURNED VALUE**

None.

## **CALLERS**

Application.

## **NOTES / WARNINGS**

The *ModemCtrl* request is described in *API for FTxxxx Devices Application Note AN\_115*, revision 1.1, section 3.2.

## C-1-4 USBH\_FTDI\_FlowCtrlSet()

Sets the flow control handshaking of the communication port.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_FlowCtrlSet (USBH_FTDI_HANDLE ftdi_handle,
                             CPU_INT08U      protocol,
                             CPU_INT08U      xon_char,
                             CPU_INT08U      xoff_char,
                             USBH_ERR        *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>protocol</code>	Protocol code. This is a bitmap of the following possible values:  <code>USBH_FTDI_PROTOCOL_RTS_CTS</code> <code>USBH_FTDI_PROTOCOL_DTR_RTS</code> <code>USBH_FTDI_PROTOCOL_XON_XOFF</code>
<code>xon_char</code>	Xon character.
<code>xoff_char</code>	Xoff character.
<code>p_err</code>	Pointer to the variable that will receive the return error code from this function.  <code>USBH_ERR_NONE</code> <code>USBH_ERR_INVALID_ARG</code> <code>USBH_ERR_DEV_NOT_READY</code> Host controller driver error code

---

## **RETURNED VALUE**

None.

## **CALLERS**

Application.

## **NOTES / WARNINGS**

The *SetFlowCtrl* request is described in *API for FTxxxx Devices Application Note AN\_115, revision 1.1, section 3.3.*

### **C-1-5 USBH\_FTDI\_BaudRateSet()**

Sets the baud rate of the communication port.

#### **FILES**

usbh\_ftdi.h/usbh\_ftdi.c

#### **PROTOTYPE**

```
void USBH_FTDI_BaudRateSet (USBH_FTDI_HANDLE ftdi_handle,  
                             CPU_INT16U      baud_rate,  
                             USBH_ERR        *p_err)
```

#### **ARGUMENTS**

**ftdi\_handle**                    Handle on FTDI device.

**baud\_rate**                    Baud rate value in bits per second.

```
USBH_FTDI_BAUD_RATE_300  
USBH_FTDI_BAUD_RATE_600  
USBH_FTDI_BAUD_RATE_1200  
USBH_FTDI_BAUD_RATE_2400  
USBH_FTDI_BAUD_RATE_4800  
USBH_FTDI_BAUD_RATE_9600  
USBH_FTDI_BAUD_RATE_19200  
USBH_FTDI_BAUD_RATE_38400  
USBH_FTDI_BAUD_RATE_57600  
USBH_FTDI_BAUD_RATE_115200  
USBH_FTDI_BAUD_RATE_230400  
USBH_FTDI_BAUD_RATE_460800  
USBH_FTDI_BAUD_RATE_921600
```

---

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_DEV\_NOT\_READY

Host controller driver error code

### **RETURNED VALUE**

None.

### **CALLERS**

Application.

### **NOTES / WARNINGS**

The *SetBaudRate* request is described in *API for FTxxxx Devices Application Note AN\_115*, revision 1.1, section 3.4.

## C-1-6 USBH\_FTDI\_DataSet()

Sets the data characteristics of the communication port.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_DataSet (USBH_FTDI_HANDLE  ftdi_handle,
                        CPU_INT08U         data_size,
                        CPU_INT08U         parity,
                        CPU_INT08U         stop_bits,
                        CPU_INT08U         break_bit,
                        USBH_ERR           *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>data_size</code>	Number of data bits.
<code>parity</code>	Parity type.  USBH_FTDI_DATA_PARITY_NONE USBH_FTDI_DATA_PARITY_ODD USBH_FTDI_DATA_PARITY_EVEN USBH_FTDI_DATA_PARITY_MARK USBH_FTDI_DATA_PARITY_SPACE
<code>stop_bits</code>	Number of stop bits.  USBH_FTDI_DATA_STOP_BITS_1 USBH_FTDI_DATA_STOP_BITS_2
<code>break_bit</code>	Break signal activation/deactivation.  USBH_FTDI_DATA_BREAK_ENABLED USBH_FTDI_DATA_BREAK_DISABLED



---

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_DEV\_NOT\_READY

Host controller driver error code

### **RETURNED VALUE**

None.

### **CALLERS**

Application.

### **NOTES / WARNINGS**

The *SetData* request is described in *API for FTxxxx Devices Application Note AN\_115*, revision 1.1, section 3.5.

## C-1-7 USBH\_FTDI\_ModemStatusGet()

Gets the current value of the serial status register.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_ModemStatusGet (USBH_FTDI_HANDLE    ftdi_handle,  
                               USBH_FTDI_SERIAL_STATUS *p_serial_status,  
                               USBH_ERR             *p_err)
```

### ARGUMENTS

**ftdi\_handle**                    Handle on FTDI device.

**p\_serial\_status**                Pointer to variable that will receive the serial status.

**p\_err**                            Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY  
Host controller driver error code

### RETURNED VALUE

None.

### CALLERS

Application.

### NOTES / WARNINGS

The *GetModemStat* request is described in *API for FTxxxx Devices Application Note AN\_115, revision 1.1, section 3.6.*

---

The device updates the serial status every 16 milliseconds.

The following masks may be used to determine if a bit of the modem status is set:

```
USBH_FTDI_MODEM_STATUS_FS
USBH_FTDI_MODEM_STATUS_HS
USBH_FTDI_MODEM_STATUS_CLEAR_TO_SEND
USBH_FTDI_MODEM_STATUS_DATA_SET_READY
USBH_FTDI_MODEM_STATUS_RING_INDICATOR
USBH_FTDI_MODEM_STATUS_DATA_CARRIER_DETECT
```

The following masks may be used to determine if a bit of the line status is set:

```
USBH_FTDI_LINE_STATUS_RX_OVERFLOW_ERROR
USBH_FTDI_LINE_STATUS_PARITY_ERROR
USBH_FTDI_LINE_STATUS_FRAMING_ERROR
USBH_FTDI_LINE_STATUS_BREAK_INTERRUPT
USBH_FTDI_LINE_STATUS_TX_REGISTER_EMPTY
USBH_FTDI_LINE_STATUS_TX_EMPTY
```

## C-1-8 USBH\_FTDI\_Tx()

Sends data to FTDI device.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
CPU_INT32U  USBH_FTDI_Tx (USBH_FTDI_HANDLE  ftdi_handle,
                        void                  *p_buf,
                        CPU_INT32U           buf_len,
                        CPU_INT16U           timeout,
                        USBH_ERR             *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>p_buf</code>	Pointer to buffer that contains the data to transfer.
<code>buf_len</code>	Buffer length in octets.
<code>timeout</code>	Timeout in milliseconds.
<code>p_err</code>	Pointer to the variable that will receive the return error code from this function.

```
USBH_ERR_NONE
USBH_ERR_NULL_PTR
USBH_ERR_INVALID_ARG
USBH_ERR_DEV_NOT_READY
USBH_ERR_OS_FAIL
USBH_ERR_OS_ABORT
USBH_ERR_OS_TIMEOUT
USBH_ERR_EP_INVALID_TYPE
USBH_ERR_EP_INVALID_STATE
Host controller driver error code
```

---

**RETURNED VALUE**

Number of octets transmitted.

**CALLERS**

Application.

**NOTES / WARNINGS**

None.

## C-1-9 USBH\_FTDI\_TxAsync()

Sends data to FTDI device. This function is non-blocking.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_TxAsync (USBH_FTDI_HANDLE    ftdi_handle,
                       void                  *p_buf,
                       CPU_INT32U           buf_len,
                       USBH_FTDI_ASYNC_TX_FNCT tx_cmpl_notify,
                       void                  *p_arg,
                       USBH_ERR             *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>p_buf</code>	Pointer to buffer that contains the data to transfer.
<code>buf_len</code>	Buffer length in octets.
<code>tx_cmpl_notify</code>	Function that will be invoked upon completion of transmit operation.
<code>p_arg</code>	Pointer to application's argument that will be passed as parameter of <code>tx_cmpl_notify</code> .
<code>p_err</code>	Pointer to the variable that will receive the return error code from this function.

```
USBH_ERR_NONE
USBH_ERR_NULL_PTR
USBH_ERR_INVALID_ARG
USBH_ERR_DEV_NOT_READY
Host controller driver error code
```

---

**RETURNED VALUE**

None.

**CALLERS**

Application.

**NOTES / WARNINGS**

None.

## C-1-10 USBH\_FTDI\_Rx()

Receives data from FTDI device.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_Rx (USBH_FTDI_HANDLE    ftdi_handle,
                  void                  *_p_buf,
                  CPU_INT32U           buf_len,
                  CPU_INT16U           timeout,
                  USBH_FTDI_SERIAL_STATUS *p_serial_status,
                  USBH_ERR              *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>p_buf</code>	Pointer to buffer that will contain the received data.
<code>buf_len</code>	Buffer length in octets.
<code>timeout</code>	Timeout in milliseconds.
<code>p_serial_status</code>	Pointer to variable that will receive the serial status.



---

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_NULL\_PTR  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY  
USBH\_ERR\_OS\_FAIL  
USBH\_ERR\_OS\_ABORT  
USBH\_ERR\_OS\_TIMEOUT  
USBH\_ERR\_EP\_INVALID\_TYPE  
USBH\_ERR\_EP\_INVALID\_STATE  
Host controller driver error code

## RETURNED VALUE

Number of octets received.

## CALLERS

Application.

## NOTES / WARNINGS

The buffer **p\_buf** will be assigned using the following pattern:

Byte 0	Byte 1	Byte 2 to buf_len
Modem Status	Line Status	Raw Data

The following masks may be used to determine if a bit of the modem status is set:

USBH\_FTDI\_MODEM\_STATUS\_FS  
USBH\_FTDI\_MODEM\_STATUS\_HS  
USBH\_FTDI\_MODEM\_STATUS\_CLEAR\_TO\_SEND  
USBH\_FTDI\_MODEM\_STATUS\_DATA\_SET\_READY  
USBH\_FTDI\_MODEM\_STATUS\_RING\_INDICATOR  
USBH\_FTDI\_MODEM\_STATUS\_DATA\_CARRIER\_DETECT

The following masks may be used to determine if a bit of the line status is set:

```
USBH_FTDI_LINE_STATUS_RX_OVERFLOW_ERROR
USBH_FTDI_LINE_STATUS_PARITY_ERROR
USBH_FTDI_LINE_STATUS_FRAMING_ERROR
USBH_FTDI_LINE_STATUS_BREAK_INTERRUPT
USBH_FTDI_LINE_STATUS_TX_REGISTER_EMPTY
USBH_FTDI_LINE_STATUS_TX_EMPTY
```

---

## C-1-11 USBH\_FTDI\_RxAsync()

Receives data from FTDI device. This function is non-blocking.

### FILES

usbh\_ftdi.h/usbh\_ftdi.c

### PROTOTYPE

```
void USBH_FTDI_RxAsync (USBH_FTDI_HANDLE    ftdi_handle,
                        void                 *p_buf,
                        CPU_INT32U          buf_len,
                        USBH_FTDI_SERIAL_STATUS *p_serial_status,
                        USBH_FTDI_ASYNC_RX_FNCT rx_cmpl_notify,
                        void                 *p_arg,
                        USBH_ERR             *p_err)
```

### ARGUMENTS

<code>ftdi_handle</code>	Handle on FTDI device.
<code>p_buf</code>	Pointer to buffer that will contain the received data.
<code>buf_len</code>	Buffer length in octets.
<code>p_serial_status</code>	Pointer to variable that will receive the serial status.
<code>rx_cmpl_notify</code>	Function that will be invoked upon completion of receive operation.
<code>p_arg</code>	Pointer to application's argument that will be passed as parameter of <code>rx_cmpl_notify</code> .

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_NULL\_PTR  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY  
Host controller driver error code

### RETURNED VALUE

None.

### CALLERS

Application.

### NOTES / WARNINGS

The buffer **p\_buf** will be assigned using the following pattern:

Byte 0	Byte 1	Byte 2 to buf_len
Modem Status	Line Status	Raw Data

The following masks may be used to determine if a bit of the modem status is set:

USBH\_FTDI\_MODEM\_STATUS\_FS  
USBH\_FTDI\_MODEM\_STATUS\_HS  
USBH\_FTDI\_MODEM\_STATUS\_CLEAR\_TO\_SEND  
USBH\_FTDI\_MODEM\_STATUS\_DATA\_SET\_READY  
USBH\_FTDI\_MODEM\_STATUS\_RING\_INDICATOR  
USBH\_FTDI\_MODEM\_STATUS\_DATA\_CARRIER\_DETECT

---

The following masks may be used to determine if a bit of the line status is set:

```
USBH_FTDI_LINE_STATUS_RX_OVERFLOW_ERROR
USBH_FTDI_LINE_STATUS_PARITY_ERROR
USBH_FTDI_LINE_STATUS_FRAMING_ERROR
USBH_FTDI_LINE_STATUS_BREAK_INTERRUPT
USBH_FTDI_LINE_STATUS_TX_REGISTER_EMPTY
USBH_FTDI_LINE_STATUS_TX_EMPTY
```



## Appendix

# D

## HID API Reference

This appendix provides a reference to the  $\mu$ C/USB-Host Human Interface Device (HID) class API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

## D-1 HID FUNCTIONS

### D-1-1 USBH\_HID\_Init()

Initializes the HID device, reads & parses the report descriptor and creates the report ID list.

#### FILES

usbh\_hid.h/usbh\_hid.c

#### PROTOTYPE

```
USBH_ERR USBH_HID_Init (USBH_HID_DEV *p_hid_dev)
```

#### ARGUMENTS

`p_hid_dev`                      Pointer to the HID device.

#### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DESC_ALLOC  
USBH_ERR_HID_RD_PARSER_FAIL  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
USBH_ERR_DESC_EXTRA_NOT_FOUND  
USBH_ERR_DESC_INVALID  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_ALLOC  
Host controller driver error code
```



---

**CALLERS**

Application.

**NOTES / WARNINGS**

None.

## D-1-2 USBH\_HID\_RefAdd()

Increments the HID device's application reference counter.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_RefAdd (USBH_HID_DEV *p_hid_dev)
```

### ARGUMENTS

`p_hid_dev`                      Pointer to the HID device.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_OS_ABORT  
USBH_ERR_OS_FAIL
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

---

### D-1-3 USBH\_HID\_RefRel()

Decrements the HID device's application reference counter. Frees the device if there is not anymore references to it.

#### FILES

usbh\_hid.h/usbh\_hid.c

#### PROTOTYPE

```
USBH_ERR USBH_HID_RefRel (USBH_HID_DEV *p_hid_dev)
```

#### ARGUMENTS

`p_hid_dev`                      Pointer to the HID device.

#### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_OS_ABORT  
USBH_ERR_OS_FAIL
```

#### CALLERS

Application.

#### NOTES / WARNINGS

None.

## D-1-4 USBH\_HID\_GetReportIDArray()

Returns the array of Report ID structures and the number of Report ID structures.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_GetReportIDArray (USBH_HID_DEV      *p_hid_dev,  
                                     USBH_HID_REPORT_ID **p_report_id,  
                                     CPU_INT08U        *p_nbr_report_id)
```

### ARGUMENTS

<code>p_hid_dev</code>	Pointer to the HID device.
<code>p_report_id</code>	Pointer to variable that will receive the array of report ID structures.
<code>p_nbr_report_id</code>	Pointer to variable that will receive the number of report ID structures.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

---

## D-1-5 USBH\_HID\_GetAppCollArray()

Returns application collection structures array and number of application collection structures.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_GetAppCollArray (USBH_HID_DEV      *p_hid_dev,  
                                   USBH_HID_APP_COLL **p_app_coll,  
                                   CPU_INT08U         *p_nbr_app_coll)
```

### ARGUMENTS

<code>p_hid_dev</code>	Pointer to the HID device.
<code>p_app_coll</code>	Pointer to variable that will receive the array of application collection structures.
<code>p_nbr_app_coll</code>	Pointer to variable that will receive the number of application collection structures.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

## D-1-6 USBH\_HID\_IsBootDev()

Tests whether the HID interface belongs to the boot subclass.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
CPU_BOOLEAN USBH_HID_IsBootDev (USBH_HID_DEV *p_hid_dev,  
                                USBH_ERR *p_err)
```

### ARGUMENTS

**p\_hid\_dev** Pointer to the HID device.

**p\_err** Pointer to the variable that will receive the return error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY

### RETURNED VALUE

**DEF\_TRUE** Device belongs to boot subclass.

**DEF\_FALSE** Otherwise.

### CALLERS

Application.

### NOTES / WARNINGS

None.

---

## D-1-7 USBH\_HID\_RxReport()

Receives input or feature report from the device. This function is blocking.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
CPU_INT08U  USBH_HID_RxReport (USBH_HID_DEV *p_hid_dev,  
                                CPU_INT08U  report_id,  
                                void          *p_buf,  
                                CPU_INT08U  buf_len,  
                                CPU_INT16U  timeout_ms,  
                                USBH_ERR    *p_err)
```

### ARGUMENTS

**p\_hid\_dev** Pointer to the HID device.

**report\_id** Report ID.

**p\_buf** Pointer to buffer that will receive the report.

**buf\_len** Buffer length, in octets.

**timeout\_ms** Timeout, in milliseconds.

**p\_err** Pointer to variable that will receive the return error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_DEV\_NOT\_READY

USBH\_ERR\_NULL\_PTR

USBH\_ERR\_EP\_INVALID\_TYPE

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_UNKNOWN

USBH\_ERR\_HC\_IO

USBH\_ERR\_EP\_STALL

Host controller driver error code

**RETURNED VALUE**

Number of octets received.

**CALLERS**

Application.

**NOTES / WARNINGS**

`p_buf` contains only the report data without the report ID.



---

## D-1-8 USBH\_HID\_TxReport()

Sends report to the device. This function is blocking.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
CPU_INT08U  USBH_HID_TxReport (USBH_HID_DEV *p_hid_dev,  
                                CPU_INT08U  report_id,  
                                void         *p_buf,  
                                CPU_INT08U  buf_len,  
                                CPU_INT16U  timeout_ms,  
                                USBH_ERR    *p_err)
```

### ARGUMENTS

**p\_hid\_dev** Pointer to the HID device.

**report\_id** Report ID.

**p\_buf** Pointer to the buffer that contains the report.

**buf\_len** Buffer length, in octets.

**timeout\_ms** Timeout, in milliseconds.

**p\_err** Pointer to variable that will receive the return error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_DEV\_NOT\_READY

USBH\_ERR\_NULL\_PTR

USBH\_ERR\_EP\_INVALID\_TYPE

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_UNKNOWN

USBH\_ERR\_HC\_IO

USBH\_ERR\_EP\_STALL

Host controller driver error code

**RETURNED VALUE**

Number of octets sent.

**CALLERS**

Application.

**NOTES / WARNINGS**

Do not add the report ID to p\_buf, it will be added automatically.

---

## D-1-9 USBH\_HID\_RegRxCB()

Registers a callback function to receive reports from the device asynchronously. The callback function will be associated to the given report ID.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_RegRxCB (USBH_HID_DEV *p_hid_dev,  
                           CPU_INT08U   report_id,  
                           USBH_HID_RXCB_FNCT async_fnct,  
                           void          *p_async_arg)
```

### ARGUMENTS

<code>p_hid_dev</code>	Pointer to the HID device.
<code>report_id</code>	Report ID.
<code>async_fnct</code>	Callback function.
<code>p_async_arg</code>	Pointer to context that will be passed to callback function.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_HID_NOT_IN_REPORT  
USBH_ERR_ALLOC  
USBH_ERR_HID_REPORT_ID
```

**CALLERS**

Application.

**NOTES / WARNINGS**

None.

---

## D-1-10 USBH\_HID\_UnregRxCB()

Unregisters the callback function for the given report ID.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_UnregRxCB (USBH_HID_DEV *p_hid_dev,  
                             CPU_INT08U   report_id)
```

### ARGUMENTS

`p_hid_dev`                      Pointer to the HID device.

`report_id`                     Report ID.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_HID_REPORTID_NOT_REGISTERED  
USBH_ERR_DEV_NOT_READY
```

### CALLERS

Application.

### NOTES / WARNINGS

None.

## D-1-11 USBH\_HID\_ProtocolSet()

Sets protocol (boot/report) of the HID device.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_ProtocolSet (USBH_HID_DEV *p_hid_dev,  
                               CPU_INT16U protocol)
```

### ARGUMENTS

`p_hid_dev` Pointer to the HID device.

`protocol` Protocol to set.

USBH\_HID\_REQ\_PROTOCOL\_BOOT  
USBH\_HID\_REQ\_PROTOCOL\_REPORT

### RETURNED VALUE

Error code from this function.

USBH\_ERR\_NONE  
USBH\_ERR\_INVALID\_ARG  
USBH\_ERR\_DEV\_NOT\_READY  
USBH\_ERR\_UNKNOWN  
USBH\_ERR\_EP\_INVALID\_STATE  
USBH\_ERR\_HC\_IO  
USBH\_ERR\_EP\_STALL  
Host controller driver error code

---

## **CALLERS**

Application.

## **NOTES / WARNINGS**

For more information on the *Set\_Protocol* request, see *Device Class Definition for Human Interface Devices (HID)*, 6/27/01, Version 1.11, section 7.2.6.

## D-1-12 USBH\_HID\_ProtocolGet()

Gets protocol (boot/report) of the HID device.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_ProtocolGet (USBH_HID_DEV *p_hid_dev,  
                               CPU_INT16U *p_protocol)
```

### ARGUMENTS

<code>p_hid_dev</code>	Pointer to the HID device.
<code>p_protocol</code>	Pointer to variable that will receive the protocol code of the device.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host controller driver error code
```

### CALLERS

Application.

### NOTES / WARNINGS

For more information on the *Get\_Protocol* request, see *Device Class Definition for Human Interface Devices (HID)*, 6/27/01, Version 1.11, section 7.2.5.



---

## D-1-13 USBH\_HID\_IdleSet()

Sets idle state duration for a given report ID.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
USBH_ERR USBH_HID_IdleSet (USBH_HID_DEV *p_hid_dev,  
                           CPU_INT08U   report_id,  
                           CPU_INT32U   dur)
```

### ARGUMENTS

<code>p_hid_dev</code>	Pointer to the HID device.
<code>report_id</code>	Report ID. If 0, idle state request will apply to all the report IDs.
<code>dur</code>	Duration of the idle state in milliseconds.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_DEV_NOT_READY  
USBH_ERR_UNKNOWN  
USBH_ERR_EP_INVALID_STATE  
USBH_ERR_HC_IO  
USBH_ERR_EP_STALL  
Host controller driver error code
```

### CALLERS

Application.

**NOTES / WARNINGS**

For more information on the *Set\_Idle* request, see *Device Class Definition for Human Interface Devices (HID)*, 6/27/01, Version 1.11, section 7.2.4.

---

## D-1-14 USBH\_HID\_IdleGet()

Gets idle duration for given report ID.

### FILES

usbh\_hid.h/usbh\_hid.c

### PROTOTYPE

```
CPU_INT32U  USBH_HID_IdleGet (USBH_HID_DEV  *p_hid_dev,  
                             CPU_INT08U    report_id,  
                             USBH_ERR      *p_err)
```

### ARGUMENTS

**p\_hid\_dev** Pointer to the HID device.

**report\_id** Report ID.

**p\_err** Pointer to variable that will receive the return error code from this function.

USBH\_ERR\_NONE

USBH\_ERR\_INVALID\_ARG

USBH\_ERR\_DEV\_NOT\_READY

USBH\_ERR\_UNKNOWN

USBH\_ERR\_EP\_INVALID\_STATE

USBH\_ERR\_HC\_IO

USBH\_ERR\_EP\_STALL

Host controller driver error code

### RETURNED VALUE

Idle duration in milliseconds.

### CALLERS

Application.

**NOTES / WARNINGS**

For more information on the *Get\_Idle* request, see *Device Class Definition for Human Interface Devices (HID)*, 6/27/01, Version 1.11, section 7.2.3.

# E

## MSC API Reference

This appendix provides a reference to the  $\mu$ C/USB-Host Mass Storage Class (MSC) API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

## **E-1 MSC FUNCTIONS**

### **E-1-1 USBH\_MSC\_RefAdd()**

Increments the MSC device's application reference counter.

#### **FILES**

usbh\_msc.h/usbh\_msc.c

#### **PROTOTYPE**

```
USBH_ERR USBH_MSC_RefAdd (USBH_MSC_DEV *p_msc_dev)
```

#### **ARGUMENTS**

`p_msc_dev`                      Pointer to the MSC device.

#### **RETURNED VALUE**

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_OS_ABORT  
USBH_ERR_OS_FAIL
```

#### **CALLERS**

Application's MSC device connection callback.

#### **NOTES / WARNINGS**

None.

---

## E-1-2 USBH\_MSC\_RefRel()

Decrements the MSC device's application reference counter. Frees the device if there is not anymore references to it.

### FILES

usbh\_msc.h/usbh\_msc.c

### PROTOTYPE

```
USBH_ERR USBH_MSC_RefRel (USBH_MSC_DEV *p_msc_dev)
```

### ARGUMENTS

`p_msc_dev`                      Pointer to the MSC device.

### RETURNED VALUE

Error code from this function.

```
USBH_ERR_NONE  
USBH_ERR_INVALID_ARG  
USBH_ERR_OS_ABORT  
USBH_ERR_OS_FAIL
```

### CALLERS

Application's MSC device disconnection callback.

### NOTES / WARNINGS

None.

## E-2 FILE SYSTEM MSC DRIVER FUNCTIONS

The functions described in this section only apply to  $\mu$ C/FS.

### E-2-1 FSDev\_MSC\_DevOpen

Adds a MSC unit.

#### FILES

fs\_dev\_msc.h/fs\_dev\_msc.c

#### PROTOTYPE

```
FS_QTY FSDev_MSC_DevOpen (USBH_MSC_DEV *p_msc_dev,  
                          FS_ERR      *p_err)
```

#### ARGUMENTS

**p\_msc\_dev**                      Pointer to the MSC device.

**p\_err**                              Pointer to variable that will receive the return error code from this function:

```
FS_ERR_NONE  
FS_ERR_NULL_PTR  
FS_ERR_DEV_UNIT_NONE_AVAIL  
FS_ERR_DEV_IO  
FS_ERR_DEV_TIMEOUT
```

#### RETURNED VALUE

Unit number to which device has been assigned.

#### CALLERS

Application's MSC device connection callback.



---

## **NOTES / WARNINGS**

The return value should be used to form the name of the volume. For example, if the return value is 4, the volume name is *msc:4*. A file name *file.txt* in the root directory of this volume would have the full path: *msc:4:\file.txt*.

## **E-2-2 FSDev\_MSC\_DevClose**

Closes a MSC unit.

### **FILES**

fs\_dev\_msc.h/fs\_dev\_msc.c

### **PROTOTYPE**

```
void FSDev_MSC_DevClose (USBH_MSC_DEV *p_msc_dev)
```

### **ARGUMENTS**

**p\_msc\_dev**                      Pointer to the MSC device.

### **RETURNED VALUE**

None.

### **CALLERS**

Application's MSC device disconnection callback.

### **NOTES / WARNINGS**

None.

# F

## Host Controller Driver API Reference

This appendix provides a reference to the Host Controller Driver (HCD) API. The following information is provided for each of the services:

- A brief description
- The function prototype
- The filename of the source code
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service

Note that since every HCD function is accessed only by function pointers via the HCD's API structure, they do not need to be globally available and should therefore be declared as `static`.

## F-1 HOST DRIVER FUNCTIONS

### F-1-1 USBH\_<controller>\_Init()

Initializes all internal variables and hardware registers necessary for host controller's driver proper operations. This function should not start the host controller or enable interrupts.

#### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

#### PROTOTYPE

```
static void USBH_<controller>_Init (USBH_HC_DRV *p_hc_drv
                                   USBH_ERR   *p_err);
```

#### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_err` Pointer to variable that will receive the return error code from this function.

#### RETURNED VALUE

None.

#### CALLERS

`USBH_HC_Add()` via '`p_hc_drv_api->Init()`'.

#### NOTES / WARNINGS

- 1 This function relies heavily on the implementation of several host board support package (BSP) functions. See “Host Driver BSP Functions” on page 292 for more information on device BSP functions.
- 2 The `Init()` function generally performs the following operations, however, depending on the host controller being initialized, functionality may need to be added or removed:

- 
- Configures clock gating to the USB host, configure all necessary I/O pins and ports, and configure the related hardware interrupts. This is generally performed via the device's BSP function pointer, `Init()`, implemented in `usbh_bsp_<controller>.c` (see section F-3-1 on page 292).
  - Resets USB controller or USB controller registers.
  - Disables and clears pending USB and root hub interrupts (should already be cleared).
  - For DMA hosts: Allocate memory for all necessary descriptors. This is performed via calls to  $\mu$ C/LIB's memory module. If memory allocation fails, set `p_err` to `USBH_ERR_ALLOC` and return.
  - Sets `p_err` to `USBH_ERR_NONE` if initialization proceeded as expected. Otherwise, set `p_err` to an appropriate error code.

## F-1-2 USBH\_<controller>\_Start()

Starts the host controller by enabling USB host controller's interrupts.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_Start (USBH_HC_DRV *p_hc_drv
                                     USBH_ERR *p_err);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_err` Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_HC_Start()` via '`p_hc_drv_api->Start()`'.

### NOTES / WARNINGS

The `Start()` function performs the following operations:

- 1 Registers driver's ISR to the interrupt vector. This is generally performed via the device's BSP function pointer, `ISR_Reg()`, implemented in `usbh_bsp_<controller>.c` (see section F-3-2 on page 293). The host's BSP `ISR_Reg()` is also responsible for enabling the host interrupt controller.
- 2 Clears all interrupt flags.
- 3 Enables interrupts on the hardware controller. The host interrupt controller should have already been configured within the host driver `Init()` function.

- 
- 4 Enables the controller.
  - 5 Sets `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, set `p_err` to an appropriate error code.

### F-1-3 USBH\_<controller>\_Stop()

Stops the host controller by disabling USB host controller's interrupts.

#### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

#### PROTOTYPE

```
static void USBH_<controller>_Stop (USBH_HC_DRV *p_hc_drv
                                   USBH_ERR *p_err);
```

#### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_err` Pointer to variable that will receive the return error code from this function.

#### RETURNED VALUE

None.

#### CALLERS

`USBH_HC_Stop()` via '`p_hc_drv_api->Stop()`'.

#### NOTES / WARNINGS

Typically, the `Stop()` function performs the following operations:

- 1 Disables the controller.
- 2 Clears and disables interrupts on the hardware device.
- 3 Unregisters the driver's ISR from the interrupt vector. This is generally performed via the device's BSP function pointer, `ISR_Unreg()`, implemented in `usbh_bsp.c` (see section F-3-3 on page 294). The host's BSP `ISR_Unreg()` is also responsible for disabling the host interrupt controller.
- 4 Sets `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, set `p_err` to an appropriate error code.



---

## F-1-4 USBH\_<controller>\_SpdGet()

Returns the speed of the host controller.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static USBH_DEV_SPD USBH_<controller>_SpdGet (USBH_HC_DRV  *p_hc_drv,  
                                              USBH_ERR      *p_err);
```

### ARGUMENTS

`p_hc_drv`     Pointer to USB host driver structure.

`p_err`        Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

Host controller speed.

USBH\_DEV\_SPD\_LOW  
USBH\_DEV\_SPD\_FULL  
USBH\_DEV\_SPD\_HIGH

### CALLERS

USBH\_HC\_Add() via 'p\_hc\_drv\_api->SpdGet()'.

### NOTES / WARNINGS

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

## **F-1-5 USBH\_<controller>\_Suspend()**

Suspends all communications on the host controller.

### **FILES**

Every host controller driver's `usbh_hcd_<controller>.c`

### **PROTOTYPE**

```
static void USBH_<controller>_Suspend (USBH_HC_DRV  *p_hc_drv,  
                                       USBH_ERR    *p_err);
```

### **ARGUMENTS**

`p_hc_drv`     Pointer to USB host driver structure.

`p_err`        Pointer to variable that will receive the return error code from this function.

### **RETURNED VALUE**

None.

### **CALLERS**

`USBH_Suspend()` via '`p_hc_drv_api->Suspend()`'.

### **NOTES / WARNINGS**

- 1 This function should suspend all the transfers on the host controller. The transfers queues (if any), should not be flushed.
- 2 The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

---

## F-1-6 USBH\_<controller>\_Resume()

Resumes all communications on the host controller.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_Resume (USBH_HC_DRV *p_hc_drv,  
                                     USBH_ERR *p_err);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_err` Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_Resume()` via `'p_hc_drv_api->Resume()'`.

### NOTES / WARNINGS

- 1 This function should resume all the transfers on the host controller.
- 2 The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

## F-1-7 USBH\_<controller>\_FrameNbrGet()

Returns the USB frame count of the host controller.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_INT32U USBH_<controller>_FrmNbrGet (USBH_HC_DRV  *p_hc_drv,  
                                               USBH_ERR      *p_err);
```

### ARGUMENTS

`p_hc_drv`     Pointer to USB host driver structure.

`p_err`        Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

USB frame count.

### CALLERS

`USBH_HC_FrameNbrGet()` via `'p_hc_drv_api->FrameNbrGet()'`.

### NOTES / WARNINGS

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

---

## F-1-8 USBH\_<controller>\_EP\_Open()

Opens and configures an endpoint given its characteristics (endpoint type, endpoint address, maximum packet size, etc).

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_EP_Open (USBH_HC_DRV *p_hc_drv,  
                                       USBH_EP      *p_ep  
                                       USBH_ERR      *p_err);
```

### ARGUMENTS

`p_hc_drv`     Pointer to USB host driver structure.

`p_ep`         Pointer to a structure describing the endpoint.

`p_err`         Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

- `USBH_EP_Open()` via `'p_hc_drv_api->EP_Open()'`
- `USBH_EP_Reset()` via `'p_hc_drv_api->EP_Open()'`
- `USBH_EP_DevDescRead()` via `'p_hc_drv_api->EP_Open()'`
- `USBH_EP_DevAddrSet()` via `'p_hc_drv_api->EP_Open()'`
- `USBH_DfltEP_Open()` via `'p_hc_drv_api->EP_Open()'`.

## **NOTES / WARNINGS**

- 1 The endpoint open function allocates the resources needed for the endpoint and endpoint management. It also configures any hardware registers related to the endpoint.
- 2 The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

---

## F-1-9 USBH\_<controller>\_EP\_Close()

Closes an endpoint, and un-initializes/clears endpoint configuration in hardware.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_EP_Close (USBH_HC_DRV  *p_hc_drv,  
                                         USBH_EP      *p_ep  
                                         USBH_ERR     *p_err);
```

### ARGUMENTS

- `p_hc_drv`    Pointer to USB host driver structure.
- `p_ep`        Pointer to a structure describing the endpoint.
- `p_err`       Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

- `USBH_EP_Close()` via `'p_hc_drv_api->EP_Close()'`
- `USBH_EP_Reset()` via `'p_hc_drv_api->EP_Close()'`
- `USBH_EP_DevDescRead()` via `'p_hc_drv_api->EP_Close()'`
- `USBH_EP_DevAddrSet()` via `'p_hc_drv_api->EP_Close()'`

### NOTES / WARNINGS

- 1 The endpoint close function frees the resources allocated to the endpoint and endpoint management. It also un-initializes/clears any hardware registers related to the endpoint.
- 2 The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

## F-1-10 USBH\_<controller>\_EP\_Abort()

Aborts any pending transfer(s) on endpoint.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_EP_Abort (USBH_HC_DRV *p_hc_drv,  
                                         USBH_EP *p_ep  
                                         USBH_ERR *p_err);
```

### ARGUMENTS

- p\_hc\_drv** Pointer to USB host driver structure.
- p\_ep** Pointer to a structure describing the endpoint.
- p\_err** Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_EP_Reset()` via `'p_hc_drv_api->EP_Abort()'`.

### NOTES / WARNINGS

- 1 The endpoint abort function flushes all pending transfers on the endpoint.
- 2 The function must set **p\_err** equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set **p\_err** to an appropriate error code.



---

## F-1-11 USBH\_<controller>\_EP\_IsHalt()

Retrieves endpoint halt (stall) status.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_EP_IsHalt (USBH_HC_DRV *p_hc_drv,  
                                                USBH_EP      *p_ep  
                                                USBH_ERR     *p_err);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_ep` Pointer to a structure describing the endpoint.

`p_err` Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

Endpoint halt status.

### CALLERS

`USBH_URB_Submit()` via `'p_hc_drv_api->EP_IsHalt()'`.

### NOTES / WARNINGS

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

## F-1-12 USBH\_<controller>\_URB\_Submit()

Submits a USB Request Block (URB) on an endpoint.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_URB_Submit (USBH_HC_DRV  *p_hc_drv,  
                                           USBH_URB      *p_urb,  
                                           USBH_ERR      *p_err);
```

### ARGUMENTS

- `p_hc_drv`     Pointer to USB host driver structure.
- `p_urb`        Pointer to a structure that represents the URB.
- `p_err`        Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_URB_Submit()` via `'p_hc_drv_api->URB_Submit()'`.

### NOTES / WARNINGS

- 1 The `URB_Submit()` function performs the following operations:
  - If host controller supports Direct Memory Access (DMA), prepares DMA descriptor and adds it to the proper endpoint transfers list.
  - If host controller does not support DMA, depending on your host controller, you may have to copy data to different hardware registers/buffers depending on the transfer type.

- 
- Signals the host controller to start the transfer by writing to the necessary hardware register(s).
  - Sets `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, sets `p_err` to an appropriate error code.

## F-1-13 USBH\_<controller>\_URB\_Complete()

Signals URB completion.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_URB_Complete (USBH_HC_DRV *p_hc_drv,  
                                             USBH_URB *p_urb,  
                                             USBH_ERR *p_err);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_urb` Pointer to a structure that represents the URB.

`p_err` Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_URB_Complete()` via `'p_hc_drv_api->URB_Complete()'`.

### NOTES / WARNINGS

1 The `URB_Complete()` function performs the following operations:

- If host controller supports DMA, frees any DMA descriptor used for the transfer.
- If host controller does not support DMA and the transfer is a data reception, copies URB's buffer to application's buffer.
- Sets the error code `p_urb->Err` of the URB depending on the transfer result. If no error occurred, sets it to `USBH_ERR_NONE`.

---

## F-1-14 USBH\_<controller>\_URB\_Abort()

Aborts a URB.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_URB_Abort (USBH_HC_DRV *p_hc_drv,  
                                         USBH_URB *p_urb,  
                                         USBH_ERR *p_err);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_urb` Pointer to a structure that represents the URB.

`p_err` Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_URB_Complete()` via `'p_hc_drv_api->URB_Abort()'`.

### NOTES / WARNINGS

1 The `URB_Abort()` function performs the following operations:

- If host controller supports DMA, frees any DMA descriptor used for the transfer.
- Sets `p_urb->Err` to `USBH_ERR_URB_ABORT`.
- Clears any hardware registers that were set for the transfer.
  - Sets `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, sets `p_err` to an appropriate error code.

## F-2 ROOT HUB DRIVER FUNCTIONS

These functions are designed to emulate typical requests that the host could make to an external hub.

### F-2-1 USBH\_<controller>\_PortStatusGet()

Retrieves status of the specified root hub port.

#### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

#### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortStatusGet (USBH_HC_DRV      *p_hc_drv,
                                                    CPU_INT08U      port_nbr,
                                                    USBH_HUB_PORT_STATUS *p_port_status);
```

#### ARGUMENTS

<code>p_hc_drv</code>	Pointer to USB host driver structure.
<code>port_nbr</code>	Root hub's port number.
<code>p_port_status</code>	Port status structure to fill.

#### RETURNED VALUE

<code>DEF_OK</code>	If operation is successful.
<code>DEF_FAIL</code>	If any error occurred.

#### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortStatusGet()'`.

---

## **NOTES / WARNINGS**

This function emulates a standard *GetPortStatus* control request. `p_port_status` structure should be filled as described in Universal Serial Bus Specification, Revision 2.0, section 11.24.2.6. Most of root hubs use hardware registers to describe the state of their port(s).

## F-2-2 USBH\_<controller>\_HubDescGet()

Retrieves root hub descriptor.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_HubDescGet (USBH_HC_DRV *p_hc_drv,
                                                  void          *p_buf,
                                                  CPU_INT08U   buf_len);
```

### ARGUMENTS

- `p_hc_drv`     Pointer to USB host driver structure.
- `p_buf`        Buffer that will contain the root hub descriptor.
- `buf_len`      Length (in octets) of the buffer.

### RETURNED VALUE

- `DEF_OK`        If operation is successful.
- `DEF_FAIL`     If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->HubDescGet()'`.

### NOTES / WARNINGS

This function emulates a standard *GetHubDescriptor* control request. `p_buf` should be filled as described in Universal Serial Bus Specification, Revision 2.0, section 11.23.2.1. The content of the root hub descriptor is usually retrieved by knowing its characteristics (using hardware data sheets, for example) and by looking at the content of some hardware registers. Fill `p_buf` up to `buf_len`. Do not return `DEF_FAIL` if `buf_len` is smaller than the normal hub descriptor length.



---

## F-2-3 USBH\_<controller>\_PortEnSet()

Sets root hub port enable.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortEnSet (USBH_HC_DRV *p_hc_drv,  
                                                CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via '`p_hc_rh_api->PortEnSet()`'.

### NOTES / WARNINGS

This function emulates a standard *SetPortFeature* `PORT_ENABLE` control request. For more information on that request, see the *Universal Serial Bus specification, revision 2.0. section 11.24.2.13 Set Port Feature*. Set necessary hardware register(s) to enable specified port.

## F-2-4 USBH\_<controller>\_PortEnClr()

Clears root hub port enable.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortEnClr (USBH_HC_DRV *p_hc_drv,  
                                                CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortEnClr()'`.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature* `PORT_ENABLE` control request. For more information on that request, see the *Universal Serial Bus specification, revision 2.0. section 11.24.2.2 Clear Port Feature*. Set necessary hardware register(s) to disable specified port.

---

## F-2-5 USBH\_<controller>\_PortEnChngClr()

Clears root hub port enable change notification.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortEnChngClr (USBH_HC_DRV *p_hc_drv,  
                                                    CPU_INT08U   port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via '`p_hc_rh_api->PortEnChngClr()`'.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature* `C_PORT_ENABLE` control request. Set necessary hardware register(s) to disable change notification of specified port enable state.

## F-2-6 USBH\_<controller>\_PortPwrSet()

Sets root hub port power.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortPwrSet (USBH_HC_DRV *p_hc_drv,  
                                                  CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortPwrSet()'`.

### NOTES / WARNINGS

This function emulates a standard *SetPortFeature* `PORT_POWER` control request. Set necessary hardware register(s) to enable power on specified port.

---

## F-2-7 USBH\_<controller>\_PortPwrClr()

Clears root hub port power.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortPwrClr (USBH_HC_DRV *p_hc_drv,  
                                                CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortPwrClr()'`.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature PORT\_POWER* control request. Set necessary hardware register(s) to disable power on specified port.

## F-2-8 USBH\_<controller>\_PortResetSet()

Sets root hub port reset state.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortResetSet (USBH_HC_DRV *p_hc_drv,  
                                                    CPU_INT08U   port_nbr);
```

### ARGUMENTS

`p_hc_drv`     Pointer to USB host driver structure.

`port_nbr`     Root hub's port number.

### RETURNED VALUE

`DEF_OK`        If operation is successful.

`DEF_FAIL`     If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via '`p_hc_rh_api->PortResetSet()`'.

### NOTES / WARNINGS

This function emulates a standard *SetPortFeature* `PORT_RESET` control request. Set necessary hardware register(s) to put specified port in the reset state.

---

## F-2-9 USBH\_<controller>\_PortResetChngClr()

Clears port reset state notification.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortResetChngClr (USBH_HC_DRV *p_hc_drv,  
                                                       CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortResetChngClr()'`.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature C\_PORT\_RESET* control request. Set necessary hardware register(s) to disable reset state change notification on specified port.

## F-2-10 USBH\_<controller>\_PortSuspendClr()

Clears root hub port suspend state.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortSuspendClr (USBH_HC_DRV *p_hc_drv,  
                                                    CPU_INT08U port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via `'p_hc_rh_api->PortSuspendClr()'`.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature* `PORT_SUSPEND` control request. Set necessary hardware register(s) to disable port suspend state on specified port.



---

## F-2-11 USBH\_<controller>\_PortConnChngClr()

Clears root hub port connection change notification.

### FILES

Every host controller driver's `usbh_hcd_<controller>.c`

### PROTOTYPE

```
static CPU_BOOLEAN USBH_<controller>_PortConnChngClr (USBH_HC_DRV *p_hc_drv,  
                                                       CPU_INT08U   port_nbr);
```

### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`port_nbr` Root hub's port number.

### RETURNED VALUE

`DEF_OK` If operation is successful.

`DEF_FAIL` If any error occurred.

### CALLERS

`USBH_HUB_RHCtrlReq()` via '`p_hc_rh_api->PortConnChngClr()`'.

### NOTES / WARNINGS

This function emulates a standard *ClearPortFeature C\_PORT\_CONNECTION* control request. Set necessary hardware register(s) to disable port connection change notification on specified port.

## **F-2-12 USBH\_<controller>\_IntEn()**

Enables root hub interrupts.

### **FILES**

Every host controller driver's `usbh_hcd_<controller>.c`

### **PROTOTYPE**

```
static CPU_BOOLEAN USBH_<controller>_IntEn (USBH_HC_DRV *p_hc_drv);
```

### **ARGUMENTS**

`p_hc_drv`     Pointer to USB host driver structure.

### **RETURNED VALUE**

`DEF_OK`        If operation is successful.

`DEF_FAIL`     If any error occurred.

### **CALLERS**

`USBH_HUB_EventReq()` via '`p_hc_rh_api->IntEn()`'.

### **NOTES / WARNINGS**

None.

---

## **F-2-13 USBH\_<controller>\_IntDis()**

Disables root hub interrupts.

### **FILES**

Every host controller driver's `usbh_hcd_<controller>.c`

### **PROTOTYPE**

```
static CPU_BOOLEAN USBH_<controller>_IntDis (USBH_HC_DRV *p_hc_drv);
```

### **ARGUMENTS**

`p_hc_drv`     Pointer to USB host driver structure.

### **RETURNED VALUE**

`DEF_OK`        If operation is successful.

`DEF_FAIL`     If any error occurred.

### **CALLERS**

`USBH_HUB_RHEvent()` via `'p_hc_rh_api->IntDis()'`.

### **NOTES / WARNINGS**

None.

## F-3 HOST DRIVER BSP FUNCTIONS

### F-3-1 USBH\_<controller>\_BSP\_Init()

Initializes board-specific USB controller dependencies.

#### FILES

Every host controller driver's `usbh_bsp_<controller>.c`

#### PROTOTYPE

```
static void USBH_<controller>_BSP_Init (USBH_HC_DRV *p_hc_drv,  
                                       USBH_ERR *p_err);
```

#### ARGUMENTS

`p_hc_drv` Pointer to USB host driver structure.

`p_err` Pointer to variable that will receive the return error code from this function.

#### RETURNED VALUE

None.

#### CALLERS

`USBH_<controller>_Init()`.

#### NOTES / WARNINGS

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

---

## F-3-2 USBH\_<controller>\_BSP\_ISR\_Reg()

Registers USB driver's ISR to interrupt vector.

### FILES

Every host controller driver's `usbh_bsp_<controller>.c`

### PROTOTYPE

```
static void USBH_<controller>_BSP_ISR_Reg (CPU_FNCT_PTR   isr_funct,  
                                           USBH_ERR       *p_err);
```

### ARGUMENTS

`isr_funct`     Pointer to driver's ISR handler.

`p_err`        Pointer to variable that will receive the return error code from this function.

### RETURNED VALUE

None.

### CALLERS

`USBH_<controller>_Start()`.

### NOTES / WARNINGS

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

### **F-3-3 USBH\_<controller>\_BSP\_ISR\_Unreg()**

Unregisters USB driver's ISR to interrupt vector.

#### **FILES**

Every host controller driver's `usbh_bsp_<controller>.c`

#### **PROTOTYPE**

```
static void USBH_<controller>_BSP_ISR_Unreg (USBH_ERR    *p_err);
```

#### **ARGUMENTS**

`p_err`            Pointer to variable that will receive the return error code from this function.

#### **RETURNED VALUE**

None.

#### **CALLERS**

`USBH_<controller>_Stop()`.

#### **NOTES / WARNINGS**

The function must set `p_err` equal to `USBH_ERR_NONE` if no errors have occurred. Otherwise, it must set `p_err` to an appropriate error code.

## Appendix

# G

## Error Codes

This appendix provides a brief explanation of  $\mu$ C/USB-Host error codes defined in `usbh_err.h`. Any error codes not listed here may be searched in `usbh_err.h` for both their numerical value and use. This appendix also contains class-specific error codes.

Each error has a numerical value. The error codes are grouped. The definition of the groups are:

<b>Error code group</b>	<b>Numbering series</b>
GENERIC	0
DEVICE	100
CONFIGURATION	200
INTERFACE	300
ENDPOINT	400
USB REQUEST BLOCK (URB)	500
DESCRIPTOR	600
HOST CONTROLLER (HC)	700
KERNEL LAYER	800
CLASS	1000
HUB CLASS	1200
HUMAN INTERFACE DEVICE (HID) CLASS	1300
MASS STORAGE CLASS (MSC)	1400
FUTURE TECHNOLOGY DEVICES INTERNATIONAL (FTDI) CLASS	1500

## G-1 GENERIC ERROR CODES

Number	Code name	Description
0	USBH_ERR_NONE	No error.
1	USBH_ERR_FAIL	Hardware error occurred.
2	USBH_ERR_ALLOC	Object/memory allocation failed.
3	USBH_ERR_FREE	Object/memory de-allocation failed.
4	USBH_ERR_INVALID_ARG	Invalid argument(s).
5	USBH_ERR_NULL_PTR	Pointer argument(s) passed NULL pointer(s).
6	USBH_ERR_BW_NOT_AVAIL	Bandwidth not available for endpoint.
7	USBH_ERR_NOT_SUPPORTED	Feature or request not supported.
8	USBH_ERR_UNKNOWN	Unknown error occurred.

## G-2 DEVICE ERROR CODES

Number	Code name	Description
100	USBH_ERR_DEV_ALLOC	Device allocation failed. Consider increasing USBH_CFG_MAX_NBR_DEVS constant.
101	USBH_ERR_DEV_NOT_READY	Device is not ready.
102	USBH_ERR_DEV_NOT_RESPONDING	Device is not responding.
103	USBH_ERR_DEV_NOT_HS	Device is not High-Speed.

## G-3 CONFIGURATION ERROR CODES

Number	Code name	Description
200	USBH_ERR_CFG_ALLOC	Configuration allocation failed. Consider increasing USBH_CFG_MAX_NBR_CFGS constant.
201	USBH_ERR_CFG_MAX_CFG_LEN	Configuration descriptor too long. Consider increasing USBH_CFG_MAX_CFG_DATA_LEN constant.



---

## G-4 INTERFACE ERROR CODES

Number	Code name	Description
300	USBH_ERR_IF_ALLOC	Interface allocation failed. Consider increasing USBH_CFG_MAX_NBR_IFS constant.

## G-5 ENDPOINT ERROR CODES

Number	Code name	Description
400	USBH_ERR_EP_ALLOC	Endpoint allocation failed. Consider increasing USBH_CFG_MAX_NBR_EPS constant.
401	USBH_ERR_EP_FREE	Endpoint de-allocation failed.
402	USBH_ERR_EP_INVALID_STATE	Endpoint is in an invalid state.
403	USBH_ERR_EP_INVALID_TYPE	Endpoint type is invalid.
404	USBH_ERR_EP_STALL	Endpoint is in a stall condition.
405	USBH_ERR_EP_NACK	Endpoint transaction returned a NAK handshake.
406	USBH_ERR_EP_NOT_FOUND	Endpoint not found.

## G-6 URB ERROR CODES

Number	Code name	Description
500	USBH_ERR_URB_ABORT	URB has been aborted.

## G-7 DESCRIPTOR ERROR CODES

Number	Code name	Description
600	USBH_ERR_DESC_ALLOC	Isochronous descriptor allocation failed. Consider increasing USBH_CFG_MAX_ISOC_DESC constant.
601	USBH_ERR_DESC_INVALID	Descriptor contains at least one invalid field.
602	USBH_ERR_DESC_LANG_ID_NOT_SUPPORTED	Language ID is not supported.
603	USBH_ERR_DESC_EXTRA_NOT_FOUND	Extra descriptor not found.

## G-8 HOST CONTROLLER ERROR CODES

Number	Code name	Description
700	USBH_ERR_HC_ALLOC	Host Controller allocation failed. Consider increasing USBH_CFG_MAX_NBR_HC constant.
701	USBH_ERR_HC_INIT	Host Controller initialization failed.
702	USBH_ERR_HC_START	Host Controller start failed.
703	USBH_ERR_HC_IO	Host Controller general Input/Output error.
704	USBH_ERR_HC_HALTED	Host Controller in halted state.
705	USBH_ERR_HC_PORT_RESET	Host Controller port reset error.

## G-9 KERNEL LAYER ERROR CODES

Number	Code name	Description
800	USBH_ERR_OS_TASK_CREATE	Task creation failed.
801	USBH_ERR_OS_SIGNAL_CREATE	Signal creation failed
802	USBH_ERR_OS_DEL	Service close failed.
803	USBH_ERR_OS_TIMEOUT	Signal pend timed-out.
804	USBH_ERR_OS_ABORT	Signal pend aborted.
805	USBH_ERR_OS_FAIL	Operation failed.

---

## G-10 CLASS ERROR CODES

Number	Code name	Description
1000	USBH_ERR_CLASS_PROBE_FAIL	Class probing failed.
1001	USBH_ERR_CLASS_DRV_NOT_FOUND	Class driver not found.
1002	USBH_ERR_CLASS_DRV_ALLOC	Class driver allocation failed. Consider increasing USBH_CFG_MAX_NBR_CLASS_DRVS constant.

## G-11 HUB CLASS ERROR CODES

Number	Code name	Description
1200	USBH_ERR_HUB_INVALID_PORT_NBR	Invalid port number.
1201	USBH_ERR_HUB_PORT	Port error.

## G-12 HUMAN INTERFACE DEVICE (HID) CLASS ERROR CODES

Number	Code name	Description
1300	USBH_ERR_HID_ITEM_LONG	Long items not supported.
1301	USBH_ERR_HID_ITEM_UNKNOWN	Unknown item type.
1302	USBH_ERR_HID_MISMATCH_COLL	Collection mismatch.
1303	USBH_ERR_HID_NOT_APP_COLL	Not an application collection.
1304	USBH_ERR_HID_REPORT_OUTSIDE_COLL	Report not part of a collection.
1305	USBH_ERR_HID_MISMATCH_PUSH_POP	Push/pop items mismatch.
1306	USBH_ERR_HID_USAGE_PAGE_INVALID	Invalid usage page.
1307	USBH_ERR_HID_REPORT_ID	Invalid report ID.
1308	USBH_ERR_HID_REPORT_CNT	Invalid report count.
1309	USBH_ERR_HID_PUSH_SIZE	Invalid push size.
1310	USBH_ERR_HID_POP_SIZE	Invalid pop size.
1311	USBH_ERR_HID_REPORT_INVALID_VAL	Report format contains invalid value(s).
1312	USBH_ERR_HID_RD_PARSER_FAIL	Report descriptor parsing failed.

<b>Number</b>	<b>Code name</b>	<b>Description</b>
1313	USBH_ERR_HID_NOT_IN_REPORT	No IN report.

### **G-13 MASS STORAGE CLASS (MSC) ERROR CODES**

<b>Number</b>	<b>Code name</b>	<b>Description</b>
1400	USBH_ERR_MSC_CMD_FAILED	CSW command error status.
1401	USBH_ERR_MSC_CMD_PHASE	CSW phase error status.
1402	USBH_ERR_MSC_IO	MSC Input/Output error.
1403	USBH_ERR_MSC_LUN_ALLOC	Logical unit allocation failed.

### **G-14 FUTURE TECHNOLOGY DEVICES INTERNATIONAL (FTDI) CLASS ERROR CODES**

<b>Number</b>	<b>Code name</b>	<b>Description</b>
1500	USBH_ERR_FTDI_LINE	Error detected on line status.