emUSB-Host

CPU independent
USB Host stack for
embedded applications

User & reference guide

Document: UM10001
Software version 1.16c
Revision: 0
Date: May 13, 2015

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the latest software version. If any error occurs, please inform us and we will try to assist you as soon as possible.

For further information on topics or routines not yet specified, please contact us.

Software	Revision	Date	By	Description
1.16c	0	150513	YR	Update to latest software version.
internal	0	150512	YR	Added FAQ Chapter. Several small improvements.
internal	0	150316	YR	Many grammatical and stylistic improvements.
1.16b	1	150209	YR	Update to latest software version.
1.16a	1	150204	SR	Chapter Configuration: * Added USBH_EHCI_Config_SetM2MEndianMode()
1.16	0	141208	YR	Chapter CDC: * Added USBH_CDC_SetConfigFlags()
internal	0	141117	YR	Several improvements to descriptions. Minor spelling corrections.
internal	0	140916	YR	Several improvements to descriptions.
1.15b	0	140829	YR	Chapter CDC: * Added USBH_CDC_ReadAsync() * Added USBH_CDC_WriteAsync() * Added USBH_CDC_RW_CONTEXT * Added USBH_CDC_ON_COMPLETE_FUNC
1.14d	0	140516	SR	Update to latest software version.
1.14c	0	140428	SR	Update to latest software version.
1.14b	0	140401	SR	Update to latest software version.
1.14a	0	140320	SR	Update to latest software version.
1.12h	0	140304	SR	Update to latest software version.
1.12g	0	140225	SR	Update to latest software version.
1.12f	0	140221	SR	Update to latest software version.
1.12e	0	140210	SR	Update to latest software version.
1.12d	0	131202	SR	Update to latest software version.
1.12c	0	131018	YR	Update to latest software version.
1.12b	0	130927	YR	Update to latest software version.
1.12a	0	130920	YR	Added EHCI controller specifics.
1.10	1	120926	SR	Chapter Host controller specifics: * Added new drivers to the list. Chapter Host controller specifics: Updated performance values.
1.10	0	120515	YR	Chapter "FT232" and chapter "CDC" added. Chapter Host controller specifics: * Added new drivers to the list.
1.08	2	111114	SR	Added new driver for Atmel AVR32. Updated cross-references. Updated Running emUSBH.
1.06	2	110905	SR	Added new chapter "CDC device driver".
1.06	1	110825	SR	Added new chapter "OnTheGo Add-On".
1.06	0	110615	SR	Added new chapter "Host controller specific" Added pictures to chapter HID, MSD, Printer Updated Configuration chapter Added Sample app chapter Added information that a RTOS is necessary Updated Information in chapter Introduction Update functions descriptions in chapter API. Added new driver configuration in chapter Configuration
1.02	0	100806	MD	Added screenshots to chapter “Running emUSB-Host on target hardware”. Renamed function parameters to conform with our coding standards. Changed the return values of HID API functions to USBH_STATUS. Added detail descriptions to example applications.
1.01	0	100721	MD	Chapter “Printer” added. Corrected various function prototypes.
1.00	0	090609	AS	Initial version.

Software versions

Refers to *Release_emUSBH.html* for information about the changes of the software versions.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that emUSB-Host offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 1.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction

This chapter provides an introduction to using emUSB-Host. It explains the basic concepts behind emUSB-Host.

1.1 What is emUSB-Host

emUSB-Host is a CPU-independent USB Host stack.

emUSB-Host is a high-performance library that has been optimized for speed, versatility and small memory footprint.

1.2 Features

emUSB-Host is written in ANSI C and can be used on virtually any CPU. Here is a list of emUSB-Host features:

- ISO/ANSI C source code.
- High performance.
- Small footprint.
- No configuration required.
- Runs out-of-the-box.
- Control, bulk and interrupt transfers.
- Very simple host controller driver structure.
- USB Mass Storage Device Class available.
- Works seamlessly with embOS and emFile (for MSD).
- Support for class drivers.
- Support for external USB hub devices.
- Support for devices with alternate settings.
- Support for multi-interface devices.
- Support for multi-configuration devices.
- Royalty-free.

1.3 Basic concepts

emUSB Host consists of three layers: a driver for hardware access, the emUSB-Host core and a USB class driver. For a functional emUSB-Host, the core component and at least one of the hardware drivers is necessary. emUSB-Host handles all USB operations independently in a seperate task(s) beside the target application task. This implicity means that an RTOS is required.
A recommendation is using embOS since it perfectly fits the requirements of emUSB Host and works seamlessly with emUSB-Host, not requiring any integration work

1.4 Tasks and interrupt usage

emUSB-Host uses two dedicated tasks. One of the tasks processes the interrupts generated by the USB host controller. The function USBH_ISRTask() should run as this task with the highest priority. The other task manages the internal software timers. Its routine should be the USBH_Task() function. The priorities of both tasks have to be higher than the priority of any other application task which uses emUSB-Host.

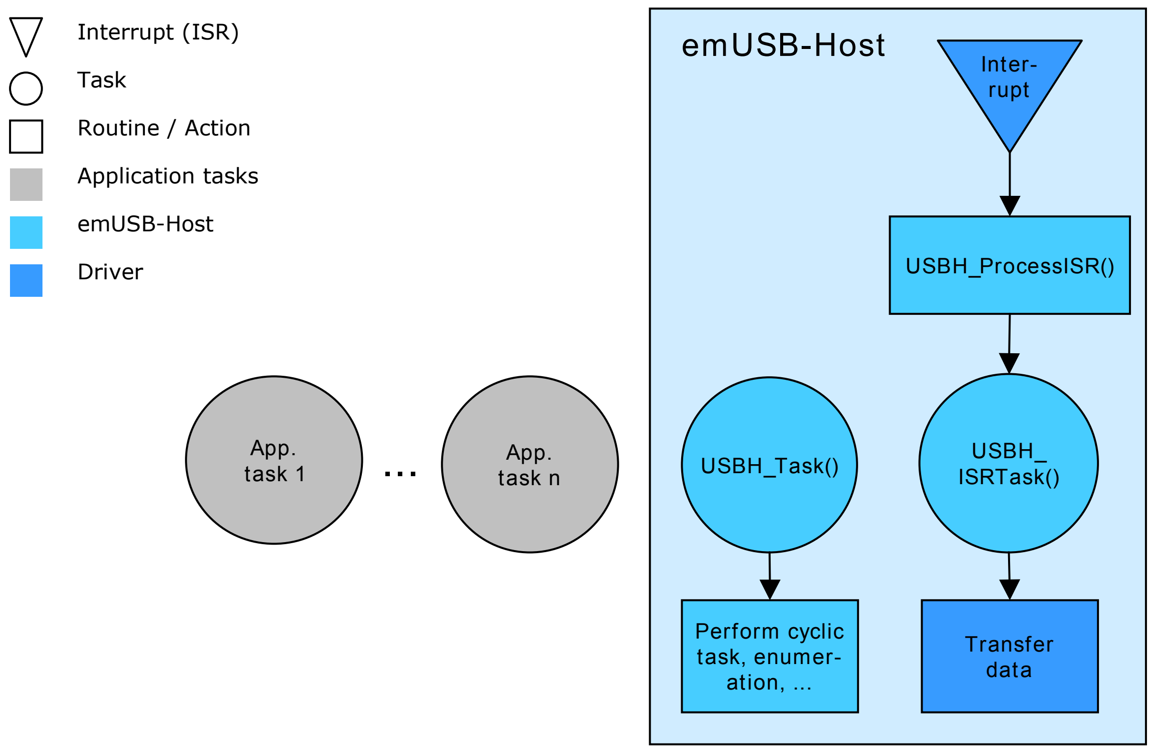
To recap:

- USBH_ISRTask runs with the highest priority
- USBH_Task runs with a priority lower than USBH_ISRTask
- All application tasks run with a priority lower than USBH_Task

Especially when using MSD it is easy to forget that the file system functions actually call emUSB-Host functions underneath. Therefore a task operating on the file system of a connected USB medium is considered an application task and should have a lower priority than USBH_Task.

Tasks which do not use emUSB-Host in any way can run at a higher priority than USBH_ISRTask. Even if a different high-priority task blocks the CPU for extended periods of time, USB communication should not be affected. USB communication is host-controlled, there are no timeouts on the device side and the host is free to delay the communication depending on how busy it is.

Your application must properly configure these two tasks at startup. The examples in the `Application` folder show how to do this.



1.5 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant C compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

1.6 Use of undocumented functions

Functions, variables and data-types which are not explained in this manual are considered internal. They are in no way required to use the software. Your application should not use and rely on any of the internal elements, as only the documented API functions are guaranteed to remain unchanged in future versions of the software.

If you feel that it is necessary to use undocumented (internal) functions, please get in touch with SEGGER support in order to find a solution.

Chapter 2

Background information

This is a short introduction to USB. The fundamentals of USB are explained and links to additional resources are given.

2.1 USB

2.1.1 Short Overview

The Universal Serial Bus (USB) is an external bus architecture for connecting peripherals to a host computer. It is an industry standard — maintained by the USB Implementers Forum — and because of its many advantages it enjoys a huge industry-wide acceptance. Over the years, a number of USB-capable peripherals appeared on the market, for example printers, keyboards, mice, digital cameras etc. Among the top benefits of USB are:

- Excellent plug-and-play capabilities allow devices to be added to the host system without reboots (“hot-plug”). Plugged-in devices are identified by the host and the appropriate drivers are loaded instantly.
- USB allows easy extensions of host systems without requiring host-internal extension cards.
- Device bandwidths may range from a few Kbytes/second to hundreds of Mbytes/second.
- A wide range of packet sizes and data transfer rates are supported.
- USB provides internal error handling. Together with the hot-plug capability mentioned before this greatly improves robustness.
- The provisions for powering connected devices dispense the need for extra power supplies for many low power devices.
- Several transfer modes are supported which ensures the wide applicability of USB.

These benefits have not only led to broad market acceptance, but have also produced several other advantages, such as low costs of USB cables and connectors or a wide range of USB stack implementations. Last but not least, the major operating systems such as Microsoft Windows XP, Mac OS X, or Linux provide excellent USB support.

2.1.2 Important USB Standard Versions

USB 1.1 (September 1998)

This standard version supports isochronous and asynchronous data transfers. It has dual speed data transfer of 1.5 Mbytes/second for low speed and 12 Mbytes/second for full speed devices. The maximum cable length between host and device is five meters. Up to 500 mA of electric current may be distributed to low power devices.

USB 2.0 (April 2000)

As all previous USB standards, USB 2.0 is fully forward and backward compatible. Existing cables and connectors may be reused. A new high speed transfer speed of 480 Mbytes/second (40 times faster tha USB 1.1 at full speed) was added.

2.1.3 USB System Architecture

A USB system is composed of three parts - a host side, a device side and a physical bus. The physical bus is represented by the USB cable and connects the host and the device.

The USB system architecture is asymmetric. Every single host can be connected to multiple devices in a tree-like fashion using special hub devices. You can connect up to 127 devices to a single host, but the count must include the hub devices as well.

USB Host

A USB host consists of a USB host controller hardware and a layered software stack. This host stack contains:

- A host controller driver (HCD) which provides the functionality of the host controller hardware.
- The USB Driver (USBD) Layer which implements the high level functions used by USB device drivers in terms of the functionality provided by the HCD.

- The USB Device drivers which establish connections to USB devices. The driver classes are also located here and provide generic access to certain types of devices such as printers or mass storage devices.

USB Device

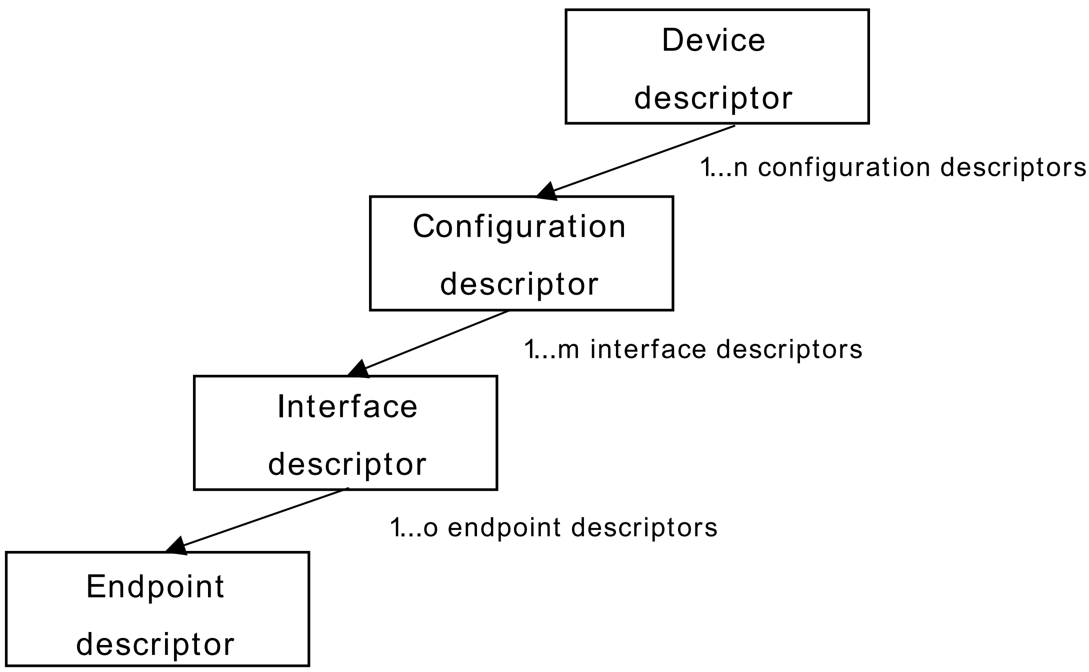
Two types of devices exist: hubs and functions. Hubs usually provide four additional USB attachment points. Functions provide capabilities to the host and are able to transmit or receive data or control information over the USB bus. Every peripheral USB device represents at least one function but may implement more than one function. A USB printer for instance may provide file system like access in addition to printing.

In this guide we treat the term USB device as synonymous with functions and will not consider hubs.

Each USB device contains configuration information which describes its capabilities and resource requirements. Before it can be used a USB device must be configured by the host. When a new device is connected for the first time, the host enumerates it, requests the configuration from the device, and performs the actual configuration. For example, if an embedded device uses emUSB-MSD, the embedded device will appear as a USB mass storage device, and the host OS provides the driver out of the box. In general, there is no need to develop a custom driver to communicate with target devices that use one of the USB class protocols.

Descriptors

A device reports its attributes via descriptors. Descriptors are data structures with a standard defined format. A USB device has one *device descriptor* which contains information applicable to the device and all of its configurations. It also contains the number of configurations the device supports. For each configuration, a *configuration descriptor* contains configuration-specific information. The configuration descriptor also contains the number of interfaces provided by the configuration. An interface groups the endpoints into logical units. Each *interface descriptor* contains information about the number of endpoints. Each endpoint has its own *endpoint descriptor* which states the endpoint's address, transfer types etc.



As can be seen, the descriptors form a tree. The root is the device descriptor with *n* configuration descriptors as children, each of which has *m* interface descriptors which in turn have *o* endpoint descriptors each.

2.1.4 Transfer Types

The USB standard defines four transfer types: control, isochronous, interrupt, and bulk. Control transfers are used in the setup phase. The application can basically select one of the other three transfer types. For most embedded applications, bulk is the best choice because it allows the highest possible data rates.

Control transfers

Typically used for configuring a device when attached to the host. It may also be used for other device-specific purposes, including control of other pipes on the device.

Isochronous transfers

Typically used for applications which need guaranteed speed. Isochronous transfer is fast but with possible data loss. A typical use is for audio data which requires a constant data rate.

Interrupt transfers

Typically used by devices that need guaranteed quick responses (bounded latency).

Bulk transfers

Typically used by devices that generate or consume data in relatively large and bursty quantities. Bulk transfer has wide dynamic latitude in transmission constraints. It can use all remaining available bandwidth, but with no guarantees on bandwidth or latency. Because the USB bus is normally not very busy, there is typically 90% or more of the bandwidth available for USB transfers.

2.1.5 Setup phase / Enumeration

The host first needs to get information from the target before the target can start communicating with the host. This information is gathered in the initial setup phase. The information is contained in the descriptors. The most important part of target device identification are the product and Vendor IDs. During the setup phase, the host also assigns an address to the client. This part of the setup is called *enumeration*.

2.1.6 Product / Vendor IDs

A Vendor ID can be obtained from the USB Implementers Forum, Inc. (*www.usb.org*). This is necessary only when development of the product is finished; during the development phase, the supplied vendor and Product IDs can be used as samples.

2.2 Predefined device classes

The USB Implementers Forum has defined device classes for different purposes. In general, every device class defines a protocol for a particular type of application such as a mass storage device (MSD), human interface device (HID), etc.

2.3 References

For additional information see the following documents:

- Universal Serial Bus Specification, Revision 2.0
- Universal Serial Bus Mass Storage Class Specification Overview, Rev 1.2
- UFI command specification: USB Mass Storage Class, UFI Command Specification, Rev 1.0

Chapter 3

Running emUSB-Host on target hardware

This chapter explains how to integrate and run emUSB-Host on your target hardware. It explains this process step-by-step.

Integrating emUSB-Host

The emUSB-Host default configuration is preconfigured with valid values, which match the requirements of most applications. emUSB-Host is designed to be used with embOS, SEGGER's real-time operating system. We recommend to start with an embOS sample project and include emUSB-Host into this project.

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. In this document the IAR Embedded Workbench IDE is used for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use makefiles; in this case, when we say "add to the project", this translates into "add to the makefile".

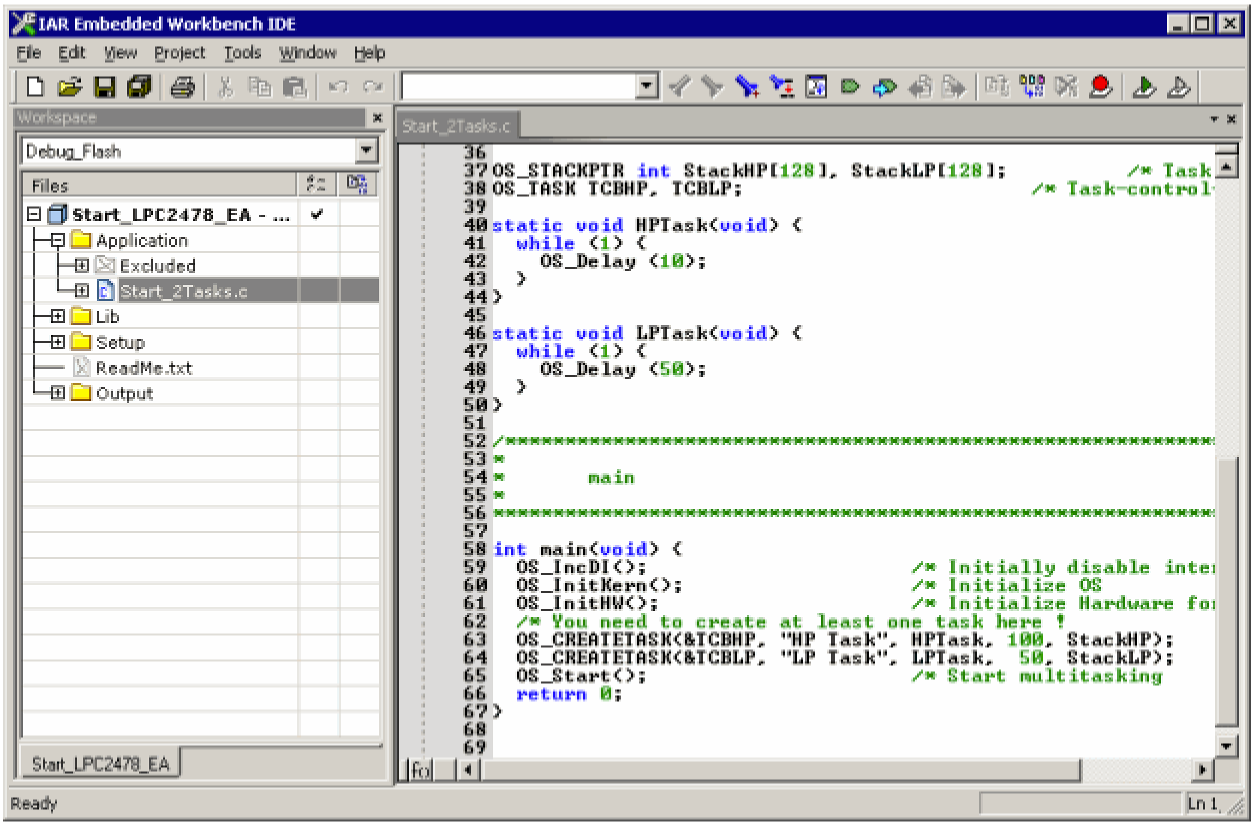
Procedure to follow

Integration of emUSB-Host is a relatively simple process, which consists of the following steps:

- Step 1: Open an emUSB-Host project and compile it.
- Step 2: Add emUSB-Host to the start project.
- Step 3: Compile the project.

3.1 Step 1: Open an embOS start project

We recommend that you use one of the supplied embOS start projects for your target system. Compile the project and run it on your target hardware.



3.2 Step 2: Adding emUSB-Host to the start project

Add all source files in the following folders to your project:

- `Config`
- `USBH`

The `Config` folder includes all configuration files of emUSB-Host. The configuration files are preconfigured with valid values, which match the requirements of most applications. Add the hardware configuration `USBH_Config_<TargetName>.c` supplied with the driver shipment.

If your hardware is currently not supported, use the example configuration file and the driver template to write your own driver. The example configuration file and the driver template is located in the `Sample\Driver\Template` folder.

The `Util` folder is an optional component of the emUSB-Host shipment. It contains optimized MCU and/or compiler specific files, for example a special *memcpy* function.

Replace BSP.c and BSP.h of your emUSB-Host start project

Replace the `BSP.c` source file and the `BSP.h` header file used in your emUSB-Host start project with the one which is supplied with the emUSB-Host shipment. If there is no `BSP.c` is available for your device/target device, either check www.segger.com whether there is an eval-package available with a BSP.c which can be used for your target device, otherwise please contact SEGGER. Some drivers require a special function which initializes the USB Host interface. This function is called `BSP_USBH_Init()`. It is used to enable the ports which are connected to the hardware. All interface driver packages include the `BSP.c` and `BSP.h` files irrespective of whether the `BSP_USBH_Init()` function is implemented or not.

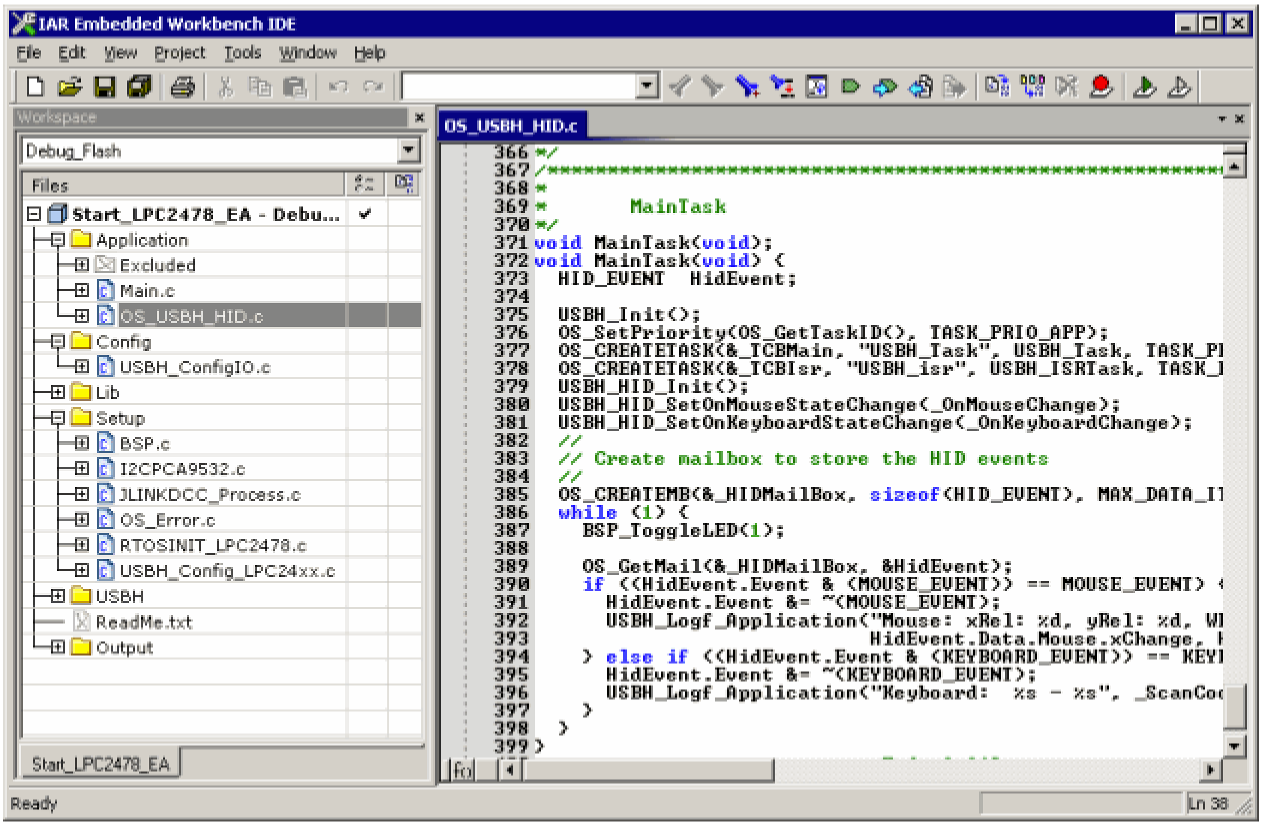
Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same folder as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `Inc`
- `USBH`

Select the start application

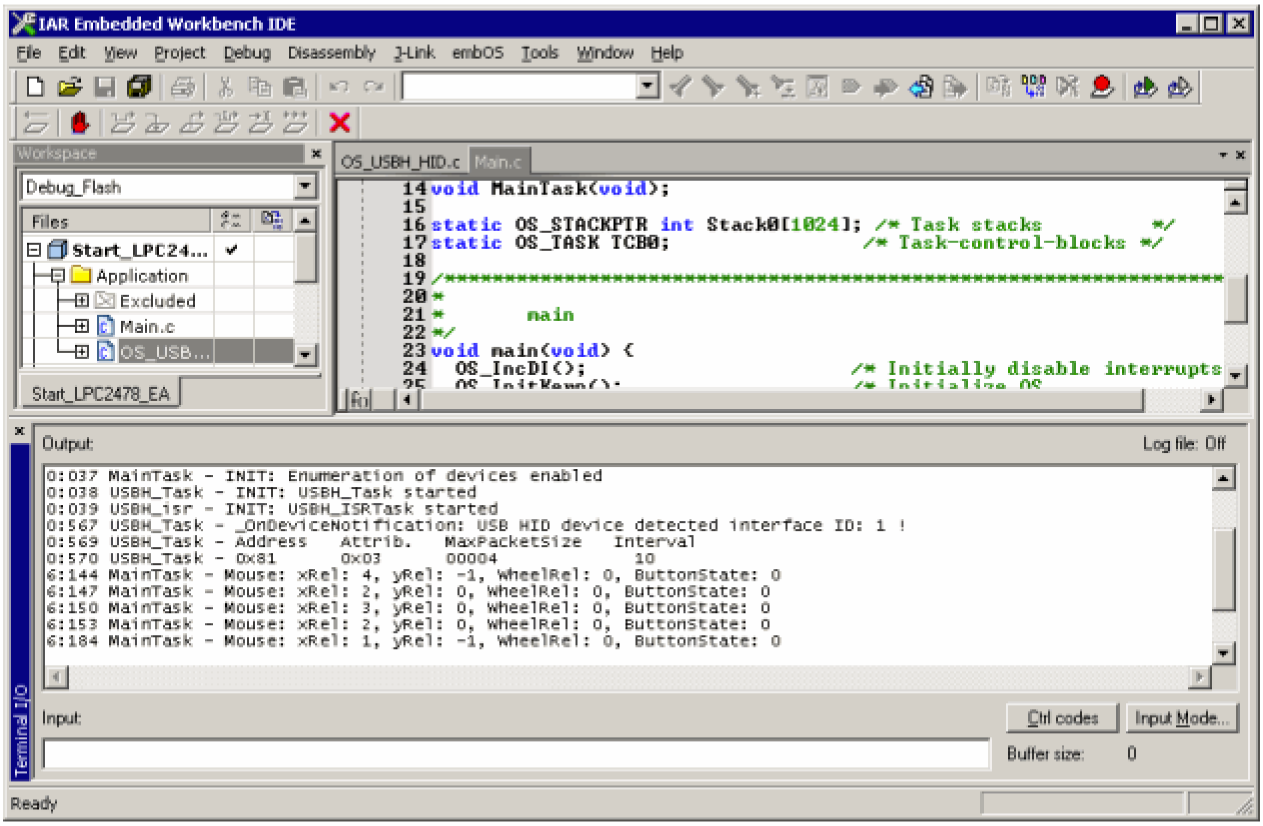
For quick and easy testing of your emUSB-Host integration, start with the code found in the `Application` folder. Add one of the applications to your project (for example `OS_USBH_HID.c`).



3.3 Step 3: Build the project and test it

Build the project. It should compile without errors and warnings. If you encounter any problem during the build process, check your include path and your project configuration settings. To test the project, download the output into your target and start the application.

The sample application waits for events generated by mice and keyboards. Simply connect a mouse or a keyboard to host and see the output generated in the terminal I/O of the debugger. A mouse will generate events when it is moved or when its buttons are pressed, as you can see in the screenshot below. A keyboard will generate events when the keys are pressed and released.



Chapter 4

Example applications

In this chapter, you will find a description of each emUSB-Host example application.

4.1 Overview

Various example applications for emUSB-Host are supplied. These can be used for testing the correct installation and proper function of the device running emUSB-Host.

The following start application files are provided:

File	Description
`OS_USBH_HID.c`	Demonstrates the handling of mouse and keyboard events.
`OS_USBH_MSD.c`	Demonstrates how to handle mass storage devices.
`OS_USBH_Printer.c`	Shows how to interact with a printer.

Table 4.1: emUSB-Host example applications

The example applications for the target-side are supplied in source code in the `Application` folder of your shipment.

4.2 Mouse and keyboard events (`OS_USBH_HID.c`)

This example application displays in the terminal I/O of the debugger the events generated by a mouse and a keyboard connected over USB.

A message in the form:

```
6:972 MainTask - Mouse: xRel: 0, yRel: 0, WheelRel: 0, ButtonState: 1
```

is generated each time the mouse generates an event. An event is generated when the mouse is moved, a button is pressed or the scroll-wheel is rolled. The message indicates the change in position over the vertical and horizontal axis, the scroll-wheel displacement and the status of all buttons.

In case of a keyboard these two messages are generated when a key is pressed and then released:

```
386:203 MainTask - Keyboard: Key e/E               - pressed
386:287 MainTask - Keyboard:  Key e/E               - released
```

The keycode is displayed followed by its status.

4.3 Mass storage handling (`OS_USBH_MSD.c`)

This demonstrates the handling of mass storage devices. A small test is run as soon as a mass storage device is connected to host. The results of the test are displayed in the terminal I/O window of the debugger. If the medium is not formatted only the message "`Medium is not formatted`." is shown and the application waits for a new device to be connected. In case the medium is formatted the file system is mounted and the total disk space is displayed. The test goes on and creates a file named `TestFile.txt` in the root directory of the disk followed by a listing of the files in the root directory. The value returned by `OS_GetTime()` is stored in the created file. At the end of test the file system is unmounted and information about the mass storage device is displayed like Vendor ID and name.

This is the information shown when a 16GB SanDisk Cruzer USB memory stick is connected:

```
**** Device added

38:127 MainTask - Running sample on "msd:0:"

38:129 MainTask -
**** Volume information for msd:0:
   0015640000 KBytes total disk space
   0014668096 KBytes avai
38:130 MainTask -
Creating file msd:0:\TestFile.txt...
38:178 MainTask - Ok

38:179 MainTask - Contents of msd:0:

38:184 MainTask - TESTFILE.TXT       Attributes: A--- Size: 20

38:188 MainTask - M (Dir) Attributes: ---- Size: 0

38:195 MainTask - A (Dir) Attributes: ---- Size: 0

38:211 MainTask -
**** Unmount ****

38:213 MainTask -
Test with following data was successful:


VendorId:       0x 781
ProductId:      0x5406
VendorName:     SanDisk
ProductName:    Cruzer
Revision:       8.02
NumSectors:     31301631
BytesPerSector: 512
TotalSize:      15283 MByte
101:593 USBH_Task -
**** Device removed
```


4.4 Printer interaction (OS_USBH_Printer.c)

This example shows how to communicate with a printer connected over USB. As soon as a printer connects over USB the message "**** Device added" is displayed on the terminal I/O window of the debugger followed by the device ID of the printer and the port status. After that the ASCII text "Hello World" and a form feed is sent to printer.

Terminal output:

```
**** Device added
Device Id  = MFG:Hewlett-Packard;CMD:PJL,PML,POSTSCRIPT,PCLXL,PCL;MDL:HP
LaserJet P2015 Series;CLS:PRINTER;DES:Hewlett-Packard LaserJet P2015
Series;MEM:MEM=23MB;COMMENT:RES=1200x1;
PortStatus = 0x18 ->NoError=1, Select/OnLine=1, PaperEmpty=0
Printing Hello World to printer
Printing completed

**** Device removed
```


Chapter 5

USB Host Core

In this chapter, you will find a description of all API functions as well as all required data and function types.

5.1 API Functions

The table below lists the available API functions. The functions are listed in alphabetical order.

Function	Description
USBH_AssignMemory()	Configures a memory pool for emUSB-Host internal handling.
USBH_AssignTransferMemory()	Configures a memory pool for the data exchange with the host controller.
USBH_CloseInterface()	Closes a previously opened interface.
USBH_CreateInterfaceList()	Generates a list of available interfaces.
USBH_DestroyInterfaceList()	Deletes a previously generated interface list.
USBH_Exit()	Is called to exit of library.
USBH_GetCurrentConfigurationDescriptor()	Retrieves the current configuration descriptor.
USBH_GetDeviceDescriptor()	Retrieves the device descriptor.
USBH_GetEndpointDescriptor()	Retrieves an endpoint descriptor.
USBH_GetFrameNumber()	Retrieves the current frame number.
USBH_GetInterfaceDescriptor()	Retrieves the interface descriptor.
USBH_GetInterfaceId()	Returns the interface Id for a specified interface.
USBH_GetInterfaceIdByHandle()	Retrieves the current frame number.
USBH_GetInterfaceInfo()	Obtains information about a specified interface.
USBH_GetSerialNumber()	Retrieves the serial number.
USBH_GetSpeed()	Retrieves the operation speed of the device.
USBH_GetStatusStr()	Return the status as a string constants.
USBH_ISRTask()	Processes the events triggered from the interrupt handler.
USBH_Init()	Initializes the emUSB-Host stack.
USBH_OpenInterface()	Opens the specified interface.
USBH_RegisterEnumErrorNotification()	Registers a port error enumeration notification.
USBH_RegisterPnPNotification()	Registers a notification function for PnP events.
USBH_RestartEnumError()	Restarts the enumerations of all failed/not recognized devices .
USBH_SubmitUrb()	Is used to submit an URB.
USBH_Task()	Manages the internal software timers.
USBH_UnregisterEnumErrorNotification()	Unregisters an registered port error enumeration notification.
USBH_UnregisterPnPNotification()	Unregisters a previously registered notification for PnP events.

Table 5.1: emUSB-Host API function overview

5.1.1 USBH_AssignMemory()

Description

Sets up storage for the memory allocator.

Prototype

```
void USBH_AssignMemory(U32 * pMem, U32 NumBytes);
```

Parameter

Parameter	Description
pMem	Pointer to a caller allocated memory area.
NumBytes	Size of memory area in bytes.

Table 5.2: USBH_AssignMemory() parameter list

Additional information

emUSB-Host comes with its own dynamic memory allocator optimized for its needs. You can use this function to set up up a memory area for the heap. The best place to call it is in the USBH_X_Config() function.

In cases where the USB host controller has limited access to system memory, the USBH_AssignTransferMemory() must be called in addition.

5.1.2 USBH_AssignTransferMemory()

Description

Sets up additional storage for the memory allocator. The USB host controller must have read/write access to the configured memory area.

Prototype

```
void USBH_AssignTransferMemory(U32 * pMem, U32 NumBytes);
```

Parameter

Parameter	Description
pMem	Pointer to a memory area.
NumBytes	Size of memory area in bytes.

Table 5.3: USBH_AssignTransferMemory() parameter list

Additional information

This function should be called from USBH_X_Config().

5.1.3 USBH_CloseInterface()

Description

Closes the specified interface.

Prototype

```
void USBH_CloseInterface(USBH_INTERFACE_HANDLE hInterface);
```

Parameter

Parameter	Description
hInterface	Contains the handle for an interface opened by a call to USBH_OpenInterface(). It must not be NULL.

Table 5.4: USBH_CloseInterface() parameter list

Additional information

Each handle must be closed one time. Calling this function with an invalid handle leads to undefined behavior.

5.1.4 USBH_CreateInterfaceList()

Description

Generates a list of available interfaces matching a given criteria.

Prototype

```
USBH_INTERFACE_LIST_HANDLE USBH_CreateInterfaceList(
                              USBH_INTERFACE_MASK    * pInterfaceMask,
                              unsigned int           * pInterfaceCount);
```

Parameters

Parameter	Description
pInterfaceMask	Pointer to a caller provided structure. IN: allows you to select interfaces to be included in the list. If this pointer is NULL all available interfaces are returned. OUT: ---
pInterfaceCount	Pointer to a caller provided counter. IN: --- OUT: Number of interfaces in the list.

Table 5.5: USBH_CreateInterfaceList() parameter list

Return value

On success it returns a handle to the interface list. In case of an error it returns NULL.

Additional information

The generated interface list is stored in the emUSB-Host and must be deleted by a call to USBH_DestroyInterfaceList(). The list contains a snapshot of interfaces available at the point of time where the function is called. This enables the application to have a fixed relation between the index and a USB interface in a list. The list is not updated if a device is removed or connected. A new list must be created to capture the current available interfaces. Hub devices are not added to the list!

Example

```
/*********************************************************************
*
*       _ListJLinkDevices
*
*   Function description
*     Generates a list of JLink devices connected to host.
*/
static void _ListJLinkDevices(void) {
  USBH_INTERFACE_MASK IfaceMask;
  unsigned int IfaceCount;
  USBH_INTERFACE_LIST_HANDLE hIfaceList;

  memset(&IfaceMask, 0, sizeof(IfaceMask));
  //
  // We want a list of all SEGGER J-Link devices connected to our host.
  // The devices are selected by their Vendor and Product ID.
  // Other identification information is not taken into account.
  //
  IfaceMask.Mask = USBH_INFO_MASK_VID | USBH_INFO_MASK_PID;
  IfaceMask.VendorId  = 0x1366;
  IfaceMask.ProductId = 0x0101;
  hIfaceList = USBH_CreateInterfaceList(&IfaceMask, &IfaceCount);
  if (hIfaceList == NULL) {
    USBH_Warnf_Application("Cannot create the interface list!");
  } else {
    if (IfaceCount == 0) {
      USBH_Logf_Application("No devices found.");
    } else {
      unsigned int i;
      USBH_INTERFACE_ID IfaceId;
      //
      // Traverse the list of devices and display information about each of them
```



```
        //
        for (i = 0; i < IfaceCount; ++i) {
          //
          // An interface is addressed by its ID
          //
          IfaceId = USBH_GetInterfaceId(hIfaceList, i);
          if (IfaceId == 0) {
            USBH_Warnf_Application("Cannot find interface with index %d!", i);
          } else {
            _ShowIfaceInfo(IfaceId);
          }
        }
      }
      //
      // Ensure the list is properly cleaned up
      //
      USBH_DestroyInterfaceList(hIfaceList);
    }
  }
```


5.1.5 USBH_DestroyInterfaceList()

Description

Deletes a previously generated interface list.

Prototype

```
void USBH_DestroyInterfaceList(
     USBH_INTERFACE_LIST_HANDLE hInterfaceList);
```

Parameter

Parameter	Description
hInterfaceList	Contains the handle to the interface list to remove. It must not be NULL.

Table 5.6: USBH_DestroyInterfaceList() parameter list

Additional information

Deletes an interface list generated by a previous call to USBH_CreateInterfaceList(). If an interface list is not deleted the library has a memory leak.

5.1.6 USBH_Exit()

Description

Is called to exit of library.

Prototype

```
void USBH_Exit();
```

Additional information

Has to be called on exit of the library. The library may free global resources within this function. This includes also the removing and deleting of added host controllers. After this function call, no other function of the library should be called.

5.1.7 USBH_GetCurrentConfigurationDescriptor()

Description

Retrieves the current configuration descriptor.

Prototype

```
USBH_STATUS USBH_GetCurrentConfigurationDescriptor(
            USBH_INTERFACE_HANDLE   hInterface,
            U8                    * pBuffer,
            unsigned int          * pBufferSize);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pBuffer	Points to a caller provided buffer. IN: --- OUT: current configuration descriptor.
pBufferSize	Points to a caller provided variable. IN: size of the buffer in bytes OUT: length of the device configuration descriptor in bytes.

Table 5.7: USBH_GetCurrentConfigurationDescriptor() parameter list

Return value

USBH_STATUS_SUCCESS: OK
USBH_STATUS_DEVICE_REMOVED Device not connected

Additional information

Returns a copy of the current configuration descriptor. The descriptor is a copy that was stored during the device enumeration.
Normally this function is initially called in order to get the first part of the configuration descriptor (9 bytes) This first part contains the size of the whole configuration descriptor.
To get the other configuration descriptors from a multi-configuration device, a URB must be submitted to the device with the function set to `USBH_FUNCTION_CONTROL_REQUEST`.

5.1.8 USBH_GetDeviceDescriptor()

Description

Retrieves the device descriptor.

Prototype

```
USBH_STATUS USBH_GetDeviceDescriptor(
            USBH_INTERFACE_HANDLE   hInterface,
            U8                    * pBuffer,
            unsigned int          * pBufferSize);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pBuffer	Points to a caller provided buffer. IN: --- OUT: device descriptor.
pBufferSize	Points to a caller provided variable. IN: size of buffer in bytes OUT: length of the device descriptor in bytes.

Table 5.8: USBH_GetDeviceDescriptor() parameter list

Return value

USBH_STATUS_SUCCESS:	OK
USBH_STATUS_DEVICE_REMOVED:	Device not connected

Additional information

Returns a copy of the device descriptor without accessing the deivce. If the buffer is smaller than the device descriptor the function returns the first part of it.

5.1.9 USBH_GetEndpointDescriptor()

Description

Retrieves an endpoint descriptor.

Prototype

```
USBH_STATUS USBH_GetEndpointDescriptor(
            USBH_INTERFACE_HANDLE      hInterface,
            U8                         AlternateSetting,
            USBH_EP_MASK             * pMask,
            U8                       * pBuffer,
            unsigned int             * pBufferSize);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
AlternateSetting	Specifies the alternate setting for the interface. The function returns endpoint descriptors that are inside the specified alternate setting.
pMask	Pointer to a caller allocated structure of type USBH_EP_MASK. IN: specifies the endpoint selection pattern. OUT: ---
pBuffer	Points to a caller provided buffer. IN: --- OUT: endpoint descriptor.
pBufferSize	Points to a caller provided variable. IN: Size of buffer in bytes OUT: length of the endpoint descriptor in bytes.

Table 5.9: USBH_GetEndpointDescriptor() parameter list

Return value

USBH_STATUS_SUCCESS:	OK
USBH_STATUS_DEVICE_REMOVED:	Device not connected
USBH_STATUS_INVALID_PARAM:	Invalid parameter passed to function

Additional information

Returns a copy of the endpoint descriptor that was captured during the enumeration. The endpoint descriptor is part of the configuration descriptor.

5.1.10 USBH_GetFrameNumber()

Description

Retrieves the current frame number.

Prototype

```
USBH_STATUS USBH_GetFrameNumber(
            USBH_INTERFACE_HANDLE     hInterface,
            U32                     * pFrameNumber);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pFrameNumber	Pointer to a caller allocated variable. IN: --- OUT: current frame number.

Table 5.10: USBH_GetFrameNumber() parameter list

Return value

USBH_STATUS_SUCCESS:	On success
USBH_STATUS_DEVICE_REMOVED:	Device was removed

Additional information

The frame number is transferred on the bus with 11 bits. This frame number is returned as a 16 or 32 bit number related to the implementation of the host controller. The last 11 bits are equal to the current frame. The frame number is increased each millisecond. The same applies to high speed. The returned frame number is related to the bus where the device is connected. The frame numbers between different host controllers can be different.

5.1.11 USBH_GetInterfaceDescriptor()

Description

Retrieves the interface descriptor.

Prototype

```
USBH_STATUS USBH_GetInterfaceDescriptor(
            USBH_INTERFACE_HANDLE   hInterface,
            U8                      AlternateSetting,
            U8                    * pBuffer,
            unsigned int          * pBufferSize);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
AlternateSetting	Specifies the alternate setting for this interface.
pBuffer	Points to a caller provided buffer. IN: --- OUT: interface descriptor.
pBufferSize	Points to a caller provided variable. IN: size of buffer in bytes OUT: length of the interface descriptor in bytes.

Table 5.11: USBH_GetInterfaceDescriptor() parameter list

Return value

USBH_STATUS_SUCCESS:	OK
USBH_STATUS_DEVICE_REMOVED	Device not connected
USBH_STATUS_INVALID_PARAM:	Invalid parameter passed to function

Additional information

Returns a copy of an interface descriptor. The interface descriptor belongs to the interface that is identified by the USBH_INTERFACE_HANDLE. If the interface has different alternate settings the interface descriptors of each alternate setting can be requested. The function returns a copy of the descriptor that was requested during the enumeration. The interface descriptor is a part of the configuration descriptor.

5.1.12 USBH_GetInterfaceId()

Description

Returns the interface Id for a specified interface.

Prototype

```
USBH_INTERFACE_ID USBH_GetInterfaceId(
                  USBH_INTERFACE_LIST_HANDLE hInterfaceList,
                  unsigned int               Index);
```

Parameters

Parameter	Description
hInterfaceList	Contains the handle for the interface list generated by a call to USBH_CreateInterfaceList().
Index	Specifies the zero based index for an interface in the list.

Table 5.12: USBH_GetInterfaceId() parameter list

Return value

On success the interface Id for the interface specified by Index is returned. If the interface index does not exist the function returns 0.

Additional information

The interface ID identifies a USB interface as long as the device is connected to the host. If the device is removed and re-connected a new interface ID is assigned. The interface ID is even valid if the interface list is deleted. The function can return an interface ID even if the device is removed between the call to the function USBH_CreateInterfaceList() and the call to this function. If this is the case, the function USBH_OpenInterface() fails.

5.1.13 USBH_GetInterfaceIdByHandle()

Description

Retrieves the interface ID for a given interface.

Prototype

```
USBH_STATUS USBH_GetInterfaceIdByHandle(
            USBH_INTERFACE_HANDLE    hInterface,
            USBH_INTERFACE_ID      * pInterfaceId);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pInterfaceId	Pointer to a caller allocated handler. IN: --- OUT: interface Id.

Table 5.13: USBH_GetInterfaceIdByHandle() parameter list

Return value

USBH_STATUS_SUCCESS:	On success
USBH_STATUS_DEVICE_REMOVED:	Device was removed

Additional information

Returns the interface Id if the handle to the interface is available. This may be useful if a Plug and Play notification is received and the application checks if it is related to a given handle. The application can avoid calls to this function if the interface Id is stored in the device context of the application.

5.1.14 USBH_GetInterfaceInfo()

Description

Obtains information about a specified interface.

Prototype

```
USBH_STATUS USBH_GetInterfaceInfo(
            USBH_INTERFACE_ID      InterfaceId,
            USBH_INTERFACE_INFO * pInterfaceInfo);
```

Parameters

Parameter	Description
InterfaceId	Id of the interface to query.
pInterfaceInfo	Pointer to a caller allocated structure. IN: --- OUT: information about interface.

Table 5.14: USBH_GetInterfaceInfo() parameter list

Return value

Returns USBH_STATUS_SUCCESS on success. If the interface belongs to a device which is no longer connected to the host USBH_STATUS_DEVICE_REMOVED is returned and `pInterfaceInfo` is not filled.

Additional information

Can be used to identify a USB interface without having to open it. More detailed information can be requested after the USB interface is opened.

5.1.15 USBH_GetSerialNumber()

Description

Retrieves the serial number.

Prototype

```
USBH_STATUS USBH_GetSerialNumber(
            USBH_INTERFACE_HANDLE   hInterface,
            U8                    * pBuffer,
            unsigned int          * pBufferSize);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pBuffer	Is a pointer to a caller provided buffer. IN: --- OUT: serial number.
pBufferSize	Points to a caller provided counter. IN: size of buffer in bytes OUT: length of the serial number in bytes

Table 5.15: USBH_GetSerialNumber() parameter list

Return value

USBH_STATUS_SUCCESS: OK
USBH_STATUS_DEVICE_REMOVED: Device not connected

Additional information

Returns the serial number as a UNICODE string in USB little endian format. Count returns the number of valid bytes. The string is not zero terminated. The returned data does not contain a USB descriptor header. The descriptor is requested with the first language Id. This string is a copy of the serial number string that was requested during the enumeration. To request other string descriptors use USBH_SubmitUrb(). If the device does not support a USB serial number string the function returns success and a length of 0.

5.1.16 USBH_GetSpeed()

Description

Retrieves the operation speed of the device.

Prototype

```
USBH_STATUS USBH_GetSpeed(
            USBH_INTERFACE_HANDLE    hInterface,
            USBH_SPEED             * pSpeed);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pSpeed	Pointer to a caller allocated variable. IN: --- OUT: operating speed of device.

Table 5.16: USBH_GetSpeed() parameter list

Return value

USBH_STATUS_SUCCESS: OK
USBH_STATUS_DEVICE_REMOVED: Device was removed

Additional information

A high speed device can operate in full or high speed mode.

5.1.17 USBH_GetStatusStr()

Description

Converts the result status into a string.

Prototype

```
const char * USBH_GetStatusStr(USBH_STATUS Status);
```

Parameter

Parameter	Description
Status	Result status to convert.

Table 5.17: USBH_GetStatusStr() parameter list

Return value

Pointer to a string which contains the result status in text form.

5.1.18 USBH_Init()

Description

Initializes the emUSB-Host stack.

Prototype

```
void USBH_Init();
```

Additional information

Has to be called one time during startup before any other function. The library initializes or allocates global resources within this function.

5.1.19 USBH_ISRTask()

Description

Processes the events triggered from the interrupt handler.

Prototype

```
void USBH_ISRTask();
```

Additional information

This function should run as a separate task. It waits for events from the interrupt handler of the host controller and processes them.

Note: In order for the emUSB-Host to work reliably, the task should have the highest priority.

Example

Example in which this function is used can be found in the `Application` folder of the emUSB-Host shipment.

5.1.20 USBH_OpenInterface()

Description

Opens the specified interface.

Prototype

```
USBH_STATUS USBH_OpenInterface(
            USBH_INTERFACE_ID        InterfaceId,
            U8                       Exclusive,
            USBH_INTERFACE_HANDLE *  phInterface);
```

Parameters

Parameter	Description
InterfaceId	Specifies the interface to open by its interface Id. The interface Id can be obtained by a call to USBH_GetInterfaceId().
Exclusive	Specifies if the interface should be opened exclusive or not. If the value is nonzero the function succeeds only if no other application has an open handle to this interface.
phInterface	Pointer to a caller allocated handle. IN: --- OUT: handle to the opened interface.

Table 5.18: USBH_OpenInterface() parameter list

Return value

Returns USBH_STATUS_SUCCESS on success. The function can fail if the device was removed or the device is opened exclusively by a different application.

Additional information

The handle returned by this function via the phInterface parameter is used by the functions that perform data transfer. The returned handle must be closed with USBH_CloseInterface() when it is no longer required.

5.1.21 USBH_RegisterEnumErrorNotification()

Description

Registers a notification for a port enumeration error.

Prototype

```
USBH_ENUM_ERROR_HANDLE USBH_RegisterEnumErrorNotification(
                       void                      * pContext,
                       USBH_ON_ENUM_ERROR_FUNC   * pfOnEnumError);
```

Parameters

Parameter	Description
pContext	Is a user defined pointer that is passed unchanged to the notification callback function.
pfOnEnumError	A pointer to a notification function of type USBH_ON_ENUM_ERROR_FUNC the library calls if a port enumeration error occurs.

Table 5.19: USBH_RegisterEnumErrorNotification() parameter list

Return value

On success a valid handle to the added notification is returned. A NULL is returned in case of an error.

Additional information

To remove the notification USBH_RestartEnumError() must be called. The pfOnEnumError callback routine is called in the context of the process where the interrupt status of a host controller is processed. It is forbidden to wait in that context.

5.1.22 USBH_RegisterPnPNotification()

Description

Registers a notification function for PnP events.

Prototype

```
USBH_NOTIFICATION_HANDLE USBH_RegisterPnPNotification(
                                   USBH_PNP_NOTIFICATION * pPnPNotification);
```

Parameter

Parameter	Description
pPnPNotification	Pointer to a caller provided structure. IN: notification information. OUT: ---

Table 5.20: USBH_RegisterPnPNotification() parameter list

Return value

On success a valid handle to the added notification is returned. A NULL is returned in case of an error.

Additional information

If a valid handle is returned, the function USBH_UnregisterPnPNotification() must be called to release the notification. An application can register any number of notifications. The user notification routine is called in the context of a notify timer that is global for all USB bus PnP notifications. If this function is called while the bus driver has already enumerated devices that match the USBH_INTERFACE_MASK the callback function passed in the USBH_PNP_NOTIFICATION structure is called for each matching interface.

5.1.23 USBH_RestartEnumError()

Description

Restarts the enumeration process for all devices that have failed to enumerate.

Prototype

```
void USBH_RestartEnumError();
```

Additional information

The bus driver retries each enumeration again until the default retry count is reached.

5.1.24 USBH_SubmitUrb()

Description

Submits an URB.

Prototype

```
USBH_STATUS USBH_SubmitUrb(
             USBH_INTERFACE_HANDLE     hInterface,
             USBH_URB                * pUrb);
```

Parameters

Parameter	Description
hInterface	Specifies the interface by its interface handle.
pUrb	Pointer to a caller allocated structure. IN: contains the URB which should be submitted. OUT: contains the submitted URB with the appropriate status and the received data if any. The storage for the URB must be permanent as long as the request is pending. The host controller can define special alignment requirements for the URB or the data transfer buffer.

Table 5.21: USBH_SubmitUrb() parameter list

Return value

The request can fail for different reasons. In that case the return value is defferent from USBH_STATUS_PENDING or USBH_STATUS_SUCCESS. If the function returns USBH_STATUS_PENDING the completion function is called later. In all other cases the completion routine is not called. If the function returns USBH_STATUS_SUCCESS, the request was processed immediately. On error the request cannot be processed.

Additional information

If the status USBH_STATUS_PENDING is returned the ownership of the URB is passed to the bus driver. The storage of the URB must not be freed nor modified as long as the ownership is assigned to the bus driver. The bus driver passes the URB back to the application by calling the completion routine. An URB that transfers data can be pending for a long time.

Please make sure that the URB is not located in the stack. Otherwise the structure may be corrupted in memory. Either use *USBH_Malloc()* or use global/static memory.

Example

In the following example the function is used to turn on the NUM LOCK, CAPS LOCK and SCROLL LOCK LEDs on a keyboard. The HID report is sent over the control endpoint. The _OnCompletion callback function is called at the end of data transfer.

```
/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
USBH_URB _Urb;
```

```
/*********************************************************************
*
*       _TurnOnKeyboardLEDs
*
*   Function description
*     Turns on  NUM LOCK, CAPS LOCK and SCROLL LOCK LEDs on a keyboard.
*
*   Parameters
*     hInterface  Handle to a HID device
*/
static void _TurnOnKeyboardLEDs(USBH_INTERFACE_HANDLE hInterface) {
  U8        LedState;

  LedState = 0x07;
  _Urb.Header.pContext       = NULL;
  _Urb.Header.Function       = USBH_FUNCTION_CONTROL_REQUEST;
  _Urb.Header.pfOnCompletion = _OnCompletion;
  _Urb.Request.ControlRequest.Setup.Type    = 0x21;
  _Urb.Request.ControlRequest.Setup.Request = 0x09;
  _Urb.Request.ControlRequest.Setup.Value   = 0x0200;
  _Urb.Request.ControlRequest.Setup.Index   = 0;
  _Urb.Request.ControlRequest.Setup.Length  = 1;
  _Urb.Request.ControlRequest.pBuffer       = &LedState;
  _Urb.Request.ControlRequest.Length        = 1;
  USBH_Submit_Urb(hInterface, &_Urb);
}
```


5.1.25 USBH_Task()

Description

Manages the internal software timers.

Prototype

```
void USBH_Task();
```

Additional information

This function should run as a separate task. It iterates over the list of active timers and invokes the registered callback functions in case the timer expired.

Example

Take a look at one of the emUSB-Host examples found in the `Application` folder of your shipment.

5.1.26 USBH_UnregisterEnumErrorNotification()

Description

Removes a registered notification for a port enumeration error.

Prototype

```
void USBH_UnregisterEnumErrorNotification(
     USBH_ENUM_ERROR_HANDLE hEnumError);
```

Parameter

Parameter	Description
hEnumError	Contains the valid handle for the notification previously returned from USBH_RegisterEnumErrorNotification().

Table 5.22: USBH_UnregisterEnumErrorNotification() parameter list

Additional information

Must be called for a port enumeration error notification that was successfully registered by a call to USBH_RegisterEnumErrorNotification().

5.1.27 USBH_UnregisterPnPNotification()

Description

Removes a previously registered notification for PnP events.

Prototype

```
void USBH_UnregisterPnPNotification(
     USBH_NOTIFICATION_HANDLE hNotification);
```

Parameter

Parameter	Description
hNotification	Contains the valid handle for a PnP notification previously registered by a call to USBH_RegisterEnumErrorNotification().

Table 5.23: USBH_UnregisterPnPNotification() parameter list

Additional information

Must be called for a PnP notification that was successfully registered by a call to USBH_RegisterEnumErrorNotification().

5.2 Data Structures

The table below lists the available data structures. The structures are listed in alphabetical order.

Structure	Description
USBH_BULK_INT_REQUEST	Used to transfer data from or to a bulk endpoint.
USBH_CONTROL_REQUEST	Used as an union member for the URB data structure.
USBH_ENDPOINT_REQUEST	Is used as a union member for the URB data structure.
USBH_ENUM_ERROR	Used as a notification parameter for the enumeration error notification function.
USBH_EP_MASK	Input parameter to get an endpoint descriptor.
USBH_HEADER	Defines the header of an URB.
USBH_INTERFACE_INFO	Contains information about a USB interface and the related device.
USBH_INTERFACE_MASK	Input parameter to create an interface list or to register a PnP notification.
USBH_ISO_FRAME	Used to define ISO transfer buffers.
USBH_ISO_REQUEST	Used to transfer data to an ISO endpoint.
USBH_PNP_NOTIFICATION	Used as an input parameter for the plug-and-play notification function.
USBH_SET_CONFIGURATION	Used as a union member for the URB data structure.
USBH_SET_INTERFACE	Used as a union member for the URB data structure.
USBH_SET_POWER_STATE	Used to set a power state.
USBH_URB	Basic structure for all asynchronous operations on the bus driver.

Table 5.24: emUSB-Host data structure overview

5.2.1 USBH_BULK_INT_REQUEST

Definition

```
typedef struct USBH_BULK_INT_REQUEST {
               U8     Endpoint;
               void * Buffer;
               U32    Length;
} USBH_BULK_INT_REQUEST;
```

Description

The buffer size can be larger than the FIFO size but a host controller implementation can define a maximum size for a buffer that can be handled with one URB. For good performance and efficiency the application should use two or more buffers.

Members

Member	Description
Endpoint	Specifies the endpoint address with direction bit.
Buffer	Pointer to a caller provided buffer.
Length	Contains the size of the buffer and returns the number of bytes transferred.

Table 5.25: USBH_BULK_INT_REQUEST() member list

5.2.2 USBH_CONTROL_REQUEST

Definition

```
typedef struct USBH_CONTROL_REQUEST {
               SETUP_PACKET   Setup;
               U8             Endpoint;
               void         * Buffer;
               U32            Length;
} USBH_CONTROL_REQUEST;
```

Description

Is used to submit a control request. A control request consists of a setup phase, an optional data phase, and a handshake phase. The data phase is limited to a length of 4096 bytes. The `Setup` data structure must be filled in properly. The length field in the `Setup` must contain the size of the `Buffer`. The caller must provide the storage for the `Buffer`.

With this request each setup packet can be submitted. Some standard requests, like *SetAddress* can be sent but would destroy the multiplexing of the bus driver. It is not allowed to set the following standard requests:

SetAddress
It is assigned by the bus driver during enumeration or USB reset.

Clear Feature Endpoint Halt
Use `USBH_FUNCTION_RESET_ENDPOINT` instead. The function `USBH_FUNCTION_RESET_ENDPOINT` resets the data toggle bit in the host controller structures.

SetConfiguration
Use `USBH_SET_CONFIGURATION` instead. The bus driver must take care on the interfaces and endpoints of a configuration. The function `USBH_SET_CONFIGURATION` updates the internal structures of the driver.

Members

Member	Description
`Setup`	Specifies the setup packet.
`Endpoint`	Specifies the endpoint address with direction bit. Use 0 for default endpoint.
`Buffer`	Pointer to a caller provided buffer, can be NULL. This buffer is used in the data phase to transfer the data. The direction of the data transfer depends from the Type field in the Setup. See the USB specification for details.
`Length`	Returns the number of bytes transferred in the data phase.

Table 5.26: USBH_CONTROL_REQUEST() member list

5.2.3 USBH_ENDPOINT_REQUEST

Definition

```
typedef struct USBH_ENDPOINT_REQUEST {
               U8 Endpoint;
} USBH_ENDPOINT_REQUEST;
```

Description

Is used with the requests `USBH_FUNCTION_RESET_ENDPOINT` and `USBH_FUNCTION_ABORT_ENDPOINT`.

Members

Member	Description
`Endpoint`	Specifies the endpoint address.

Table 5.27: USBH_ENDPOINT_REQUEST() member list

5.2.4 USBH_ENUM_ERROR

Definition

```
typedef struct USBH_ENUM_ERROR {
                int         Flags;
                int         PortNumber;
                USBH_STATUS Status;
                int         ExtendedErrorInformation;
} USBH_ENUM_ERROR;
```

Description

Is used as a notification parameter for the USBH_ON_ENUM_ERROR_FUNC callback function. This data structure does not contain detailed information about the device that fails at enumeration because this information is not available in all phases of the enumeration.

Members

Member	Description
Flags	Additional flags to determine the location and the type of the error. • USBH_ENUM_ERROR_EXTHUBPORT_FLAG means the device is connected to an external hub. • USBH_ENUM_ERROR_RETRY_FLAG the bus driver retries the enumeration of this device automatically. • USBH_ENUM_ERROR_STOP_ENUM_FLAG the bus driver does not restart the enumeration for this device because all retries have failed. The application can force the bus driver to restart the enumeration by calling the function USBH_RestartEnumError. • USBH_ENUM_ERROR_DISCONNECT_FLAG means the device has been disconnected during the enumeration. If the hub port reports a disconnect state the device cannot be re-enumerated by the bus driver automatically. Also the function USBH_RestartEnumError cannot re-enumerate the device. • USBH_ENUM_ERROR_ROOT_PORT_RESET means an error during the USB reset of a root hub port occurs. • USBH_ENUM_ERROR_HUB_PORT_RESET means an error during a reset of an external hub port occurs. • UDB_ENUM_ERROR_INIT_DEVICE means an error during the device initialization (e.g. no answer to a descriptor request or it failed other standard requests). • UDB_ENUM_ERROR_INIT_HUB means the enumeration of an external hub fails.
PortNumber	Port number of the parent port where the USB device is connected. A flag in the PortFlags field determines if this is an external hub port.
Status	Status of the failed operation.
ExtendedErrorInformation	Internal information used for debugging.

Table 5.28: USBH_ENUM_ERROR() member list

5.2.5 USBH_EP_MASK

Definition

```
typedef struct USBH_EP_MASK {
               U32 Mask;
               U8  Index;
               U8  Address;
               U8  Type;
               U8  Direction;
} USBH_EP_MASK;
```

Description

Is used as an input parameter to get an endpoint descriptor. The comparison with the mask is true if each member that is marked as valid by a flag in the mask member is equal to the value stored in the endpoint. E.g. if the mask is 0 the first endpoint is returned. If `Mask` is set to `USBH_EP_MASK_INDEX` the zero based index can be used to address all endpoints.

Members

Member	Description
Mask	This member contains the information which fields are valid. It is an or'ed combination of the following flags: • `USBH_EP_MASK_INDEX` The Index is used for comparison. • `USBH_EP_MASK_ADDRESS` The Address field is used for comparison. • `USBH_EP_MASK_TYPE` The Type field is used for comparison. • `USBH_EP_MASK_DIRECTION` The Direction field is used for comparison.
Index	If valid, this member contains the zero based index of the endpoint in the interface.
Address	If valid, this member contains an endpoint address with direction bit.
Type	If valid, this member specifies a direction. It is one of the following values: • `USB_IN_DIRECTION` From device to host • `USB_OUT_DIRECTION` From host to device

Table 5.29: USBH_EP_MASK() member list

5.2.6 USBH_HEADER

Definition

```
typedef struct USBH_HEADER {
                USBH_FUNCTION                Function;
                USBH_STATUS                  Status;
                USBH_ON_COMPLETION_FUNC    * pfOnCompletion;
                void                       * pContext;
                DLIST                        ListEntry;
} USBH_HEADER;
```

Description

All members of this structure not described here are for internal use only. Do not use these members. A caller must fill in the members Function, Completion, and if required Context.

Members

Member	Description
Function	Describes the function of the request.
Status	After completion this member contains the status for the request.
pfOnCompletion	Caller provided pointer to the completion function. This completion function is called if the function USBH_SubmitUrb() returns USBH_STATUS_PENDING. If a different status code is returned the completion function is never called.
pContext	Can be used by the caller to store a context for the completion routine. It is not changed by the library.
ListEntry	Can be used to link the URB in a list. The owner of the URB can use this list entry. If the URB is passed to the library this member is used by the library.

Table 5.30: USBH_HEADER member list

5.2.7 USBH_INTERFACE_INFO

Definition

```
typedef struct USBH_INTERFACE_INFO {
               USBH_INTERFACE_ID InterfaceId;
               USB_DEVICE_ID     DeviceId;
               U16               VendorId;
               U16               ProductId;
               U16               bcdDevice;
               U8                Interface;
               U8                Class;
               U8                SubClass;
               U8                Protocol;
               unsigned int      OpenCount;
               U8                ExclusiveUsed;
               USB_SPEED         Speed;
               U8                acSerialNumber[256];
               U8                SerialNumberSize;
} USBH_INTERFACE_INFO;
```

Description

Describes the information returned by the function `USBH_GetInterfaceInfo()`.

Members

Member	Description
`InterfaceId`	Contains the unique interface Id. This Id is assigned if the USB device was successful enumerated. It is valid until the device is removed for the host. If the device is reconnected a different interface Id is assigned to each interface.
`DeviceId`	Contains the unique device Id. This Id is assigned if the USB device was successfully enumerated. It is valid until the device is removed from the host. If the device is reconnected a different device Id is assigned. The relation between the device Id and the interface Id can be used by an application to detect which USB interfaces belong to a device.
`VendorId`	Contains the Vendor ID.
`ProductId`	Contains the Product ID.
`bcdDevice`	Contains the BCD coded device version.
`Interface`	Contains the USB interface number.
`Class`	Specifies the interface class.
`Subclass`	Specifies the interface sub class.
`Protocol`	Specifies the interface protocol.
`OpenCount`	Specifies the number of open handles for this interface.
`ExclusiveUsed`	Determines if this interface is used exclusive.
`Speed`	Specifies the operation speed of this interface.
`acSerialNumber`	Contains the serial number as a counted UNICODE string.
`SerialNumberSize`	Contains the length of the serial number in bytes.

Table 5.31: USBH_INTERFACE_INFO member list

5.2.8 USBH_INTERFACE_MASK

Definition

```
typedef struct USBH_INTERFACE_MASK {
               U16 Mask;
               U16 VendorId;
               U16 ProductId;
               U16 bcdDevice;
               U8  Interface;
               U8  Class;
               U8  SubClass;
               U8  Protocol;
} USBH_INTERFACE_MASK;
```

Description

Input parameter to create an interface list or to register a PnP notification.

Members

Member	Description
Mask	Contains an or'ed selection of the following flags. If the flag is set the related member of this structure is compared to the properties of the USB interface. • USBH_INFO_MASK_VID Compare the Vendor ID (VID) of the device. • USBH_INFO_MASK_PID Compare the Product ID (PID) of the device. • USBH_INFO_MASK_DEVICE Compare the bcdDevice value of the device. • USBH_INFO_MASK_INTERFACE Compare the interface number. • USBH_INFO_MASK_CLASS Compare the class of the interface. • USBH_INFO_MASK_SUBCLASS Compare the sub class of the interface. • USBH_INFO_MASK_PROTOCOL Compare the protocol of the interface.
VendorId	Vendor ID to compare with.
ProductId	Product ID to compare with.
bcdDevice	BCD coded device version to compare with.
Interface	Interface number to compare with.
Class	Class code to compare with.
Subclass	Sub class code to compare with.
Protocol	Protocol stored in the interface to compare with.

Table 5.32: USBH_INTERFACE_MASK member list

5.2.9 USBH_ISO_FRAME

Definition

```
typedef struct USBH_ISO_FRAME {
               U32         Offset;
               U32         Length;
               USBH_STATUS Status;
} USBH_ISO_FRAME;
```

Description

Is part of USBH_ISO_REQUEST. It describes the amount of data that is transferred in one frame.

Members

Member	Description
Offset	Specifies the offset in bytes relative to the beginning of the transfer buffer.
Length	Contains the length that should be transferred in one frame.
Status	Contains the status of the operation in this frame. For an OUT endpoint this status is always success. For an IN point a CRC or Data Toggle error can be reported.

Table 5.33: USBH_ISO_FRAME() member list

5.2.10 USBH_ISO_REQUEST

Definition

```
typedef struct USBH_ISO_REQUEST{
               U8              Endpoint;
               void          * Buffer;
               U32             Length;
               unsigned int    Flags;
               unsigned int    StartFrame;
               unsigned int    Frames;
} USBH_ISO_REQUEST;
```

Description

Is not completely defined. That means the data structure consists of this data structure and an array of data structures USBH_ISO_FRAME. The size of the array is defined by Frames. Use the macro USBH_GET_ISO_URB_SIZE to get the size for an isochronous URB.

Members

Member	Description
Endpoint	Specifies the endpoint address with direction bit.
Buffer	Is a pointer to a caller provided buffer.
Length	On input this member specifies the size of the user provided buffer. On output it contains the number of bytes transferred.
Flags	This parameter contains 0 or the following flag: USBH_ISO_ASAP If this flag is set the transfer starts as soon as possible and the parameter StartFrame is ignored.
StartFrame	If the flag USBH_ISO_ASAP is not set this parameter StartFrame defines the start frame of the transfer. The StartFrame must be in the future. Use USBH_GetFrameNumber to get the current frame number. Add a time to the current frame number.
Frames	Contains the number of frames that are described with this structure.

Table 5.34: USBH_ISO_REQUEST() member list

5.2.11 USBH_PNP_NOTIFICATION

Definition

```
typedef struct USBH_PNP_NOTIFICATION {
               USBH_PnpNotification   * pfPnpNotification;
               void                   * pContext;
               USBH_INTERFACE_MASK      InterfaceMask;
} USBH_PNP_NOTIFICATION;
```

Description

Is used as an input parameter for the `USBH_RegisterEnumErrorNotification()` function.

Members

Member	Description
`PnpNotification`	Contains the notification function that is called from the library if a PnP event occurs.
`Context`	Contains the notification context that is passed unchanged to the notification function.
`InterfaceMask`	Contains a mask for the interfaces for which the PnP notification should be called.

Table 5.35: USBH_PNP_NOTIFICATION member list

5.2.12 USBH_SET_CONFIGURATION

Definition

```
typedef struct USBH_SET_CONFIGURATION {
               U8 ConfigurationDescriptorIndex;
} USBH_SET_CONFIGURATION;
```

Description

Is used with the request `USBH_FUNCTION_SET_CONFIGURATION`.

Members

Member	Description
ConfigurationDescriptorIndex	Specifies the index in the configuration description.

Table 5.36: USBH_SET_CONFIGURATION() member list

5.2.13 USBH_SET_INTERFACE

Definition

```
typedef struct USBH_SET_INTERFACE {
               U8 AlternateSetting;
} USBH_SET_INTERFACE;
```

Description

Is used with the request `USBH_FUNCTION_SET_INTERFACE`.

Members

Member	Description
AlternateSetting	Specifies the alternate setting.

Table 5.37: USBH_SET_INTERFACE() member list

5.2.14 USBH_SET_POWER_STATE

Definition

```
typedef struct USBH_SET_POWER_STATE {
               USBH_POWER_STATE PowerState;
} USBH_SET_POWER_STATE;
```

Description

If the device is switched to suspend, there must be no pending requests on the device.

Members

Member	Description
PowerState	Specifies the power state.

Table 5.38: USBH_SET_POWER_STATE() member list

5.2.15 USBH_URB

Definition

```
typedef struct USBH_URB {
               USBH_HEADER                Header;
               union {
                 USBH_CONTROL_REQUEST     ControlRequest;
                 USBH_BULK_INT_REQUEST    BulkIntRequest;
                 USBH_ISO_REQUEST         IsoRequest;
                 USBH_ENDPOINT_REQUEST    EndpointRequest;
                 USBH_SET_CONFIGURATION   SetConfiguration;
                 USBH_SET_INTERFACE       SetInterface;
                 USBH_SET_POWER_STATE     SetPowerState;
               } Request;
} USBH_URB;
```

Description

The URB is the basic structure for all asynchronous operations on the bus driver. All requests that exchange data with the device are using this data structure. The caller has to provide the memory for this structure. The memory must be permanent until the completion function is called. This data structure is used to submit an URB.

Members

Member	Description
Header	Contains the URB header of type USBH_HEADER. The most important parameters are the function code and the callback function.
Request	Is a union and contains information depending on the specific request of the USBH_HEADER.

Table 5.39: USBH_URB member list

5.3 Enumerations

The table below lists the available enumerations. The enumerations are listed in alphabetical order.

Structure	Description
USBH_DEVICE_EVENT	Enumerates the device events.
USBH_FUNCTION	Is used as a member for the USBH_HEADER data structure.
USBH_PNP_EVENT	Is used as a parameter for the PnP notification.
USBH_POWER_STATE	Enumerates the power states of a device.
USBH_SPEED	Is used to get the operation speed of a device.

Table 5.40: emUSB-Host enumerations overview

5.3.1 USBH_DEVICE_EVENT

Definition

```
typedef enum USBH_DEVICE_EVENT {
             USBH_DEVICE_EVENT_ADD,
             USBH_DEVICE_EVENT_REMOVE
} USBH_DEVICE_EVENT;
```

Description

Enumerates the types of device events. It is used by the USBH_NOTIFICATION_FUNC callback to indicate which type of event occurred.

Entries

Entry	Description
USBH_ADD_DEVICE	Indicates that a device was connected to the host and new interface is available.
USBH_REMOVE_DEVICE	Indicates that a device has been removed.

Table 5.41: USBH_DEVICE_EVENT member list

5.3.2 USBH_FUNCTION

Definition

```
typedef enum USBH_FUNCTION {
             USBH_FUNCTION_CONTROL_REQUEST,
             USBH_FUNCTION_BULK_REQUEST,
             USBH_FUNCTION_INT_REQUEST,
             USBH_FUNCTION_ISO_REQUEST,
             USBH_FUNCTION_RESET_DEVICE,
             USBH_FUNCTION_RESET_ENDPOINT,
             USBH_FUNCTION_ABORT_ENDPOINT,
             USBH_FUNCTION_SET_CONFIGURATION,
             USBH_FUNCTION_SET_INTERFACE,
             USBH_FUNCTION_SET_POWER_STATE
} USBH_FUNCTION;
```

Description

Is used as a member for the USBH_HEADER data structure. All function codes use the API function USBH_SubmitUrb() and are handled asynchronously.

Entries

Entry	Description
USBH_FUNCTION_CONTROL_REQUEST	Is used to send an URB with a control request. It uses the data structure USBH_CONTROL_REQUEST. A control request includes standard, class and vendor defines requests. The standard requests SetConfiguration, SetAddress and SetInterface cannot be submitted by this request. These requests require a special handling in the driver. See USBH_FUNCTION_SET_CONFIGURATION and USBH_FUNCTION_SET_INTERFACE for details.
USBH_FUNCTION_BULK_REQUEST	Is used to transfer data to or from a bulk endpoint. It uses the data structure USBH_BULK_INT_REQUEST.
USBH_FUNCTION_INT_REQUEST	Is used to transfer data to or from an interrupt endpoint. It uses the data structure USBH_BULK_INT_REQUEST. The interval is defined by the endpoint descriptor.
USBH_FUNCTION_ISO_REQUEST	Is used to transfer data to or from an ISO endpoint. It uses the data structure USBH_ISO_REQUEST. ISO transfer may not be supported by all host controllers.

Table 5.42: USBH_FUNCTION member list

Entry	Description
USBH_FUNCTION_RESET_DEVICE	Sends a USB reset to the device. This triggers a remove event for all interfaces of the device. After the device is successfully enumerated an arrival event is indicated. All interfaces get new interface Id's. This request uses only the URB header. If the driver indicates a device arrival event the device is in a defined state because it is reseted and enumerated by the bus driver. This request can be part of an error recovery or part of special class protocols like DFU. The application should abort all pending requests and close all handles to this device. All handles become invalid.
USBH_FUNCTION_RESET_ENDPOINT	Clears an error condition on a special endpoint. If a data transfer error occurs that cannot be handled in hardware the bus driver stops the endpoint and does not allow further data transfers before the endpoint is reset with this function. On a bulk or interrupt endpoint the host driver sends a Clear Feature Endpoint Halt request. This informs the device about the hardware error. The driver resets the data toggle bit for this endpoint. This request expects that no pending URBs are scheduled on this endpoint. Pending URBs must be aborted with the URB based function USBH_FUNCTION_ABORT_ENDPOINT. This function uses the data structure USBH_ENDPOINT_REQUEST.
USBH_FUNCTION_ABORT_ENDPOINT	Aborts all pending requests on an endpoint. The host controller calls the completion function with a status code USBH_STATUS_CANCELED. The completion of the URBs may be delayed. The application should wait until all pending requests have been returned by the driver before the handle is closed or USBH_FUNCTION_RESET_ENDPOINT is called.

Table 5.42: USBH_FUNCTION member list

Entry	Description
USBH_FUNCTION_SET_CONFIGURATION	The driver selects the configuration defined by the configuration descriptor with the index 0 during the enumeration. If the application uses this configuration there is no need to call this function. If the application wants to activate a different configuration this function must be called.
USBH_FUNCTION_SET_INTERFACE	Selects a new alternate setting for the interface. There must be no pending requests on any endpoint to this interface. The interface handle does not become invalid during this operation. The number of endpoints may be changed. This request uses the data structure USBH_SET_INTERFACE.
USBH_FUNCTION_SET_POWER_STATE	Is used to set the power state for a device. There must be no pending requests for this device if the device is set to the suspend state. The request uses the data structure USBH_SET_POWER_STATE. After the enumeration the device is in normal power state.

Table 5.42: USBH_FUNCTION member list

5.3.3 USBH_PNP_EVENT

Definition

```
typedef enum USBH_PNP_EVENT {
             USBH_ADD_DEVICE,
             USBH_REMOVE_DEVICE
} USBH_PNP_EVENT;
```

Description

Is used as a parameter for the PnP notification.

Entries

Entry	Description
USBH_ADD_DEVICE	Indicates that a device was connected to the host and a new interface is available.
USBH_REMOVE_DEVICE	Indicates that a device has been removed.

Table 5.43: USBH_PNP_EVENT member list

5.3.4 USBH_POWER_STATE

Definition

```
typedef enum USBH_POWER_STATE {
             USBH_NORMAL_POWER,
             USBH_SUSPEND
} USBH_POWER_STATE;
```

Description

Is used as a member in the USBH_SET_POWER_STATE data structure.

Entriess

Entry	Description
USBH_NORMAL_POWER	The device is switched to normal operation.
USBH_SUSPEND	The device is switched to USB suspend mode.

Table 5.44: USBH_POWER_STATE member list

5.3.5 USBH_SPEED

Definition

```
typedef enum USBH_SPEED {
             USBH_SPEED_UNKNOWN,
             USBH_LOW_SPEED,
             USBH_FULL_SPEED,
             USBH_HIGH_SPEED
} USBH_SPEED;
```

Description

Is used as a member in the USBH_INTERFACE_INFO data structure and to get the operation speed of a device.

Entries

Entry	Description
USBH_SPEED_UNKNOWN	The speed is unknown.
USBH_LOW_SPEED	The device operates at low speed.
USBH_FULL_SPEED	The device operates at full speed.
USBH_HIGH_SPEED	The device operates at high speed.

Table 5.45: USBH_SPEED member list

5.4 Function Types

The table below lists the available function types. The function types are listed in alphabetical order.

Type	Description
USBH_NOTIFICATION_FUNC	Type of callback set in USBH_PRINTER_RegisterNotification() and USBH_HID_RegisterNotification().
USBH_ON_COMPLETION_FUNC	Is called by the library when an URB request completes.
USBH_ON_ENUM_ERROR_FUNC	Is called by the library if an error occurs at enumeration stage.
USBH_ON_PNP_EVENT_FUNC	Is called by the library if a PnP event occurs and if a PnP notification was registered.

Table 5.46: emUSB-Host function type overview

5.4.1 USBH_NOTIFICATION_FUNC

Definition

```
typedef void USBH_NOTIFICATION_FUNC(
               void                   * pContext,
               U8                       DevIndex,
               USBH_DEVICE_EVENT        Event);
```

Description

This is the type of function called when a new device is added or removed.

Parameters

Parameter	Description
pContext	Pointer to a context passed by the user in the call to USBH_PRINTER_RegisterNotification() or USBH_HID_RegisterNotification() functions.
DevIndex	Zero based index of the device that was added or removed. First device has index 0, second one has index 1, etc.
Event	Gives information about the event that occurred.

Table 5.47: USBH_NOTIFICATION_FUNC() parameter list

5.4.2 USBH_ON_COMPLETION_FUNC

Definition

```
typedef void USBH_ON_COMPLETION_FUNC(USBH_URB * pUrb);
```

Description

Is called in the context of the `USBH_Task()` or `USBH_ISRTask()` functions of a host controller when an URB request finishes.

Parameter

Parameter	Description
`pUrb`	Contains the URB that was completed.

Table 5.48: USBH_ON_COMPLETION_FUNC parameter list

5.4.3 USBH_ON_ENUM_ERROR_FUNC

Definition

```
typedef void USBH_ON_ENUM_ERROR_FUNC(
             void                      * pContext,
             const USBH_ENUM_ERROR     * pEnumError);
```

Description

Is called in the context of USBH_Task() function or of a ProcessInterrupt function of a host controller. Before this function is called it must be registered with USBH_RegisterEnumErrorNotification(). If a device is not successfully enumerated the function USBH_RestartEnumError() can be called to re-start a new enumeration in the context of this function. This callback mechanism is part of the enhanced error recovery. In an embedded system with internal components connected via USB a central application may turn off the power supply for some device to force a reboot or to create an alert.

Parameters

Parameter	Description
pContext	Is a user defined pointer that was passed to USBH_RegisterEnumErrorNotification().
pEnumError	Specifies the enumeration error. This pointer is temporary and must not be accessed after the functions returns.

Table 5.49: USBH_ON_ENUM_ERROR_FUNC parameter list

5.4.4 USBH_ON_PNP_EVENT_FUNC

Definition

```
typedef void USBH_ON_PNP_EVENT_FUNC(
                void                * pContext,
                USBH_PNP_EVENT        Event,
                USBH_INTERFACE_ID     InterfaceId);
```

Description

Is called in the context of USBH_Task() function. In the context of this function all other API functions of the bus driver can be called. The removed or added interface can be identified by the interface Id. The client can use this information to find the related USB Interface and close all handles if it was in use, to open it or to collect information about the interface.

Parameters

Parameter	Description
pContext	Is the user defined pointer that was passed to USBH_RegisterEnumErrorNotification(). The library does not dereference this pointer.
Event	Specifies the PnP event.
InterfaceId	Contains the interface Id of the removed or added interface.

Table 5.50: USBH_ON_PNP_EVENT_FUNC parameter list

5.5 Error Codes

This chapter describes the error codes which are defined in the USBH.h header file.

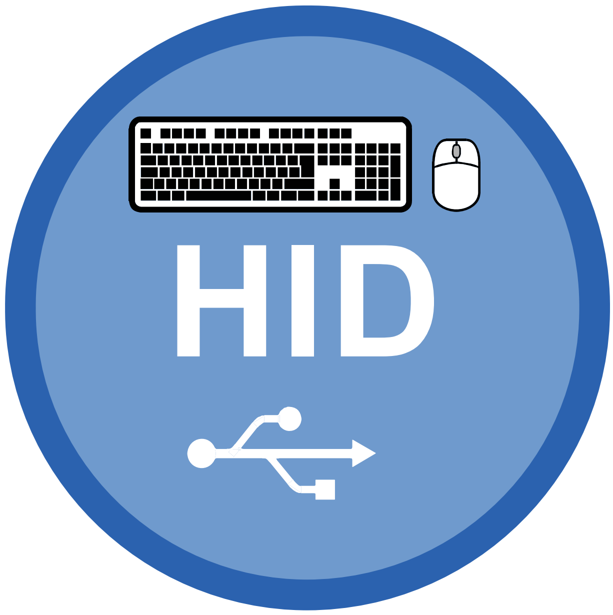
Error Code	Description
Common	
USBH_STATUS_ERROR	The operation has been completed with an error.
USBH_STATUS_INVALID_PARAM	A parameter is incorrect.
USBH_STATUS_TIMEOUT	The timeout of the operation has expired. This error code is used in all layers.
MSD specific	
USBH_STATUS_COMMAND_FAILED	This error is reported if the command code was sent successfully but the status returned from the device indicates a command error.
USBH_STATUS_INTERFACE_PROTOCOL	The used interface protocol is not supported. The interface protocol is defined by the interface descriptor.
USBH_STATUS_INTERFACE_SUB_CLASS	The used interface sub class is not supported. The interface sub class is defined by the interface descriptor.
USBH_STATUS_LENGTH	The operation detected a length error.
USBH_STATUS_SENSE_STOP	This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library will not repeat the command.
USBH_STATUS_SENSE_REPEAT	This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library repeats the command after detection of the sense code.
USBH_STATUS_WRITE_PROTECT	This error indicates that the medium is write protected. It will be returned by USBH_MSD_WriteSectors() if writing to the medium is not allowed.

Table 5.51: emUSB-Host MSD error code overview

Chapter 6

Human Interface Device HID class

This chapter describes the emUSB-Host Human interface device class driver and its usage.
The HID class is part of the Core package. The HID-class code is linked in only if registered by the application program.



6.1 Introduction

The emUSB-Host HID class software allows accessing USB Human Interface Devices.

It implements the USB Human interface Device class protocols specified by the USB Implementers Forum. The entire API of this class driver is prefixed with the "USBH_HID_" text.

This chapter describes the architecture, the features and the programming interface of this software component.

6.1.1 Overview

Two types of HIDs are currently supported: Keyboard and Mouse. For both, the application can set a callback routine which is invoked whenever a message from either one is received.

Types of HIDs:

- "True" HIDs: Mouse & Keyboard
- HID for data

6.1.2 Example code

Example code which is provided in the `OS_USBH_HID.c` file. It outputs mouse and keyboard events to the terminal I/O of debugger.

6.2 API Functions

This chapter describes the emUSB-Host HID API functions. These functions are defined in the header file USBH.h.

Function	Description
USBH_HID_CancelIo()	Interrupts a pending report read/write operation.
USBH_HID_Close()	Closes a handle to opened HID device.
USBH_HID_Exit()	Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.
USBH_HID_GetDeviceInfo()	Returns information about a HID device.
USBH_HID_GetNumDevices()	Returns the number of available HID devices and information about them.
USBH_HID_GetReport()	Reads report data from a HID device.
USBH_HID_GetReportDescriptor()	Returns the data of a report descriptor in raw form.
USBH_HID_GetReportDescriptorParsed()	Interprets the report descriptors read from a HID device.
USBH_HID_Init()	Initializes the USBH HID library.
USBH_HID_Open()	Opens a handler to a HID device by its name.
USBH_HID_OpenByIndex()	Opens a handler to a HID device by its index.
USBH_HID_RegisterNotification()	Registers a callback for the device attach/remove events.
USBH_HID_SetOnKeyboardStateChange()	Set function to be called in case of keyboard events.
USBH_HID_SetOnMouseStateChange()	Set function to be called in case of mouse events.
USBH_HID_SetReport()	Writes report data to a HID device.

Table 6.1: emUSB-Host HID API function overview

6.2.1 USBH_HID_CancelIo()

Description

Cancels a pending read/write report operation.

Prototype

```
USBH_STATUS USBH_HID_CancelIo(USBH_HID_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened HID device.

Table 6.2: USBH_HID_CancelIo() parameter list

Return value

`== USBH_STATUS_SUCCESS:`	Operation cancelled
`!= USBH_STATUS_SUCCESS:`	Error.

Additional information

You can call this function to interrupt a pending report read/write operation started by a call to `USBH_HID_GetReport()` or `USBH_HID_SetReport()` functions.

6.2.2 USBH_HID_Close()

Description

Closes a handle to an opened HID device.

Prototype

```
USBH_STATUS USBH_HID_Close(USBH_HID_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened HID device.

Table 6.3: USBH_HID_Close() parameter list

Return value

== USBH_STATUS_SUCCESS:	Device handle closed
!= USBH_STATUS_ERROR:	An error occurred

6.2.3 USBH_HID_Exit()

Description

Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.

Prototype

```
void USBH_HID_Exit(void);
```

Additional information

Has to be called if the application is closed before the USB bus driver is closed.

6.2.4 USBH_HID_GetDeviceInfo()

Description

Returns information about a connected HID device.

Prototype

```
USBH_STATUS USBH_HID_GetDeviceInfo(
            USBH_HID_HANDLE        hDevice,
            USBH_HID_DEVICE_INFO * pDevInfo);
```

Parameters

Parameter	Description
hDevice	Handle to an opened device.
pDevInfo	Pointer to a caller allocated USBH_HID_DEVICE_INFO structure. IN: --- OUT: device information.

Table 6.4: USBH_HID_GetDeviceInfo() parameter list

Return value

USBH_STATUS_INVALID_DESCRIPTOR: The report descriptor could not be parsed
USBH_STATUS_MEMORY: Not enough memory
USBH_STATUS_INVALID_PARAM: Invalid handle
USBH_STATUS_SUCCESS: Device info read

6.2.5 USBH_HID_GetNumDevices()

Description

Returns the number of available HID devices. It also retrieves the information about the connected devices.

Prototype

```
int USBH_HID_GetNumDevices(
    USBH_HID_DEVICE_INFO * pDevInfo,
    U32                    NumItems);
```

Parameters

Parameter	Description
pDevInfo	Pointer to a caller allocated array of USBH_HID_DEVICE_INFO structures. IN: --- OUT: device information.
NumItems	Number of entries in the pDevInfo array.

Table 6.5: USBH_HID_GetNumDevices() parameter list

Return value

Number of devices available.

6.2.6 USBH_HID_GetReport()

Description

Reads a report from a HID device.

Prototype

```
USBH_STATUS USBH_HID_GetReport(
            USBH_HID_HANDLE      hDevice,
            U8                 * pBuffer,
            U32                  BufferSize,
            USBH_HID_USER_FUNC * pfFunc,
            void               * pContext);
```

Parameters

Parameter	Description
hDevice	Handle to an opened device.
pBuffer	Pointer to a user allocated buffer. IN: --- OUT: data of the report descriptor in raw form.
BufferSize	Number of bytes to read.
pfFunc	Callback function of type USBH_HID_USER_FUNC invoked when the read operation finishes. It can be the NULL pointer. For further information see the "Additional information" section below.
pContext	Application specific pointer. The pointer is not dereferenced by the emUSB-Host. It is passed to the pfFunc callback function. Any value the application chooses is permitted, including NULL.

Table 6.6: USBH_HID_GetReport() parameter list

Return value

== USBH_STATUS_PENDING:	Request was submitted and application is informed via callback
== USBH_STATUS_INVALID_PARAM:	Invalid handle was passed
== USBH_STATUS_SUCCESS:	Report read

Any other value means error

Additional information

This function behaves differently whether the pfFunc points to a callback function or it is the NULL pointer.

The read operation is asynchronous if pfFunc != NULL. In other words the function returns before the data is read from the device. The emUSB-Host invokes the pfFunc callback function, in the context of the USBH_Task() routine, when the read operation ends. The read data is returned by emUSB-Host directly in the pReport array so ensure the memory location pReport points to is valid until the callback is invoked.

If the pfFunc is NULL the read operation is synchronous. That means that the function blocks until an answer is received from the device.

You can stop the read operation at any time by calling the USBH_HID_CancelIo() function.

6.2.7 USBH_HID_GetReportDescriptor()

Description

Returns the data of a report descriptor in raw form.

Prototype

```
USBH_STATUS USBH_HID_GetReportDescriptor(
            USBH_HID_HANDLE   hDevice,
            U8              * pBuffer,
            unsigned          BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to an opened device.
pBuffer	Pointer to a caller allocated buffer. IN: --- OUT: data of the report descriptor in raw form.
BufferSize	Size of buffer in bytes.

Table 6.7: USBH_HID_GetReportDescriptor() parameter list

Return value

== USBH_STATUS_SUCCESS:	Report descriptor read
== USBH_STATUS_INVALID_PARAM:	Invalid handle

6.2.8 USBH_HID_GetReportDescriptorParsed()

Description

Interprets the report descriptors read from a HID device.

Prototype

```
USBH_STATUS USBH_HID_GetReportDescriptorParsed(
            USBH_HID_HANDLE         hDevice,
            USBH_HID_REPORT_INFO  * pReportInfo,
            unsigned              * pNumEntries);
```

Parameters

Parameter	Description
hDevice	Handle to the opened HID device.
pReportInfo	Pointer to a caller allocated array of USBH_HID_REPORT_INFO structures. IN: --- OUT: parsed report data.
pNumEntries	Pointer to a caller allocated variable. IN: number of entries in the pReportInfo array OUT: number of reports available.

Table 6.8: USBH_HID_GetReportDescriptorParsed() parameter list

Return value

==USBH_STATUS_INVALID_DESCRIPTOR:	The report descriptor could not be parsed
== USBH_STATUS_MEMORY:	Insufficient memory
== USBH_STATUS_INVALID_PARAM:	Invalid handle was passed
== USBH_STATUS_SUCCESS:	Report read

Additional information

This function temporarily uses memory from the pool configured by a call of USBH_AssignMemory() function. The number of bytes allocated during the parsing depends on the number of report descriptors the HID device uses. Upon function return allocated memory is freed.

6.2.9 USBH_HID_Init()

Description

Initializes the USBH HID library.

Prototype

```
USBH_STATUS USBH_HID_Init();
```

Return value

`== USBH_STATUS_SUCCESS:`	HID component initialized
`== USBH_STATUS_ERROR:`	An error occurred

Additional information

Performs basic initialization of the library. Has to be called before any other function of the HID component is called.

6.2.10 USBH_HID_Open()

Description

Opens a handle to a HID device. The device is identified by its name.

Prototype

```
USBH_HID_HANDLE USBH_HID_Open(const char * sName);
```

Parameters

Parameter	Description
sName	Name of the HID device to open. It has the form hid*nnn* where *nnn* is 000 for the device with index 0, 001 for the device with index 1 and so on.

Table 6.9: USBH_HID_Open() parameter list

Return value

`!= 0`: Handle to a HID device
`== 0`: Device not available

Additional Information

It is recommended to use `USBH_HID_OpenByIndex()` function, since the function is faster.

6.2.11 USBH_HID_OpenByIndex()

Description

Opens a handle to a HID device. The device is identified by its index.

Prototype

```
USBH_HID_HANDLE USBH_HID_OpenByIndex(U16 Index);
```

Parameters

Parameter	Description
Index	Index of the HID device to open. The first device has the index 0, the second index 1 and so on.

Table 6.10: USBH_HID_OpenByIndex() parameter list

Return value

`!= 0:` Handle to a HID device
`== 0:` Device not available

Additional Information

The index of a HID is assigned automatically by the emUSB-Host. It remains the same as long as the HID is connected. The smallest available index is assigned to a HID at connection time.

6.2.12 USBH_HID_RegisterNotification()

Description

Registers a function the emUSB-Host should call when a HID device is attached/removed.

Prototype

```
void USBH_HID_RegisterNotification(
     USBH_NOTIFICATION_FUNC * pfFunc,
     void                   * pContext);
```

Parameters

Parameter	Description
pfFunc	Pointer to a callback function of type USBH_NOTIFICATION_FUNC the emUSB-Host calls when a HID device is attached/removed.
pContext	Application specific pointer. The pointer is not dereferenced by the emUSB-Host. It is passed to the pfFunc callback function. Any value the application chooses is permitted, including NULL.

Table 6.11: USBH_HID_RegisterNotification() parameter list

6.2.13 USBH_HID_SetOnKeyboardStateChange()

Description

Set function to be called in case of keyboard events.

Prototype

```
void USBH_HID_SetOnKeyboardStateChange(
     USBH_HID_ON_KEYBOARD_FUNC * pfOnChange);
```

Parameters

Parameter	Description
pfOnChange	Pointer to a function the library should call when a keyboard event occurs.

Table 6.12: USBH_HID_SetOnKeyboardStateChange() parameter list

Example

```
/*********************************************************************
*
*       _OnKeyboardChange
*/
static void _OnKeyboardChange(USBH_HID_KEYBOARD_DATA  * pKeyData) {
  _KeyData = *pKeyData;
  _EventOccurred |= KEYBOARD_EVENT;
  OS_EVENT_Pulse(&_Event);
}
```


6.2.14 USBH_HID_SetOnMouseStateChange()

Description

Set function to be called in case of mouse events.

Prototype

```
void USBH_HID_SetOnMouseStateChange(USBH_HID_ON_MOUSE_FUNC * pfOnChange);
```

Parameters

Parameter	Description
pfOnChange	Pointer to a function the library should call when a mouse event occurs.

Table 6.13: USBH_HID_SetOnMouseStateChange() parameter list

Example

This example shows how to handle the mouse data in the callback function. This function is called in the context of the USBH_ISRTask() and for this reason is not allowed to block for long periods of time. To overcome this, we save the data delivered by mouse in a static variable and let another task process it.

```
/*********************************************************************
*
*       _OnMouseChange
*/
static void _OnMouseChange(USBH_HID_MOUSE_DATA * pMouseData) {
  //
  // Copy into static buffer used by the task which handles Mouse input
  //
  _MouseData       = *pMouseData;
  //
  // Tell task that Mouse data is present
  //
  _EventOccurred |= MOUSE_EVENT;
  //
  // Wake the Mouse task
  //
  OS_EVENT_Pulse(&_Event);
}
```

The next example demonstrates how to interpret the data delivered by a mouse. All the work is done in the callback function. We use here a printf()-like function to show the mouse information in a human-readable form over the terminal I/O of a debugger. Note that the usage of this function in this context is discouraged. Please note that this function may block for a long period of time which would negatively affect the real-time responsiveness of emUSB-Host.

```
/*********************************************************************
*
*       _OnMouseChange
*/
static void _OnMouseChange(USBH_HID_MOUSE_DATA * pMouseData) {
  USBH_Logf_Application("xRel: %d, yRel: %d, WheelRel: %d, ButtonState: %x",
                        pMouseData->xChange,
                        pMouseData->yChange,
                        pMouseData->WheelChange,
                        pMouseData->ButtonState);
}
```

Here is a sample of the output generated when the mouse is moved:

```
15:640 USBH_isr - xRel: 4, yRel: 0, WheelRel: 0, ButtonState: 0
15:649 USBH_isr - xRel: 3, yRel: -2, WheelRel: 0, ButtonState: 0
15:659 USBH_isr - xRel: 22, yRel: -5, WheelRel: 0, ButtonState: 0
15:662 USBH_isr - xRel: 20, yRel: -1, WheelRel: 0, ButtonState: 0
15:666 USBH_isr - xRel: 4, yRel: 0, WheelRel: 0, ButtonState: 0
15:679 USBH_isr - xRel: 0, yRel: -1, WheelRel: 0, ButtonState: 0
15:682 USBH_isr - xRel: -2, yRel: 0, WheelRel: 0, ButtonState: 0
15:685 USBH_isr - xRel: -12, yRel: 0, WheelRel: 0, ButtonState: 0
```


6.2.15 USBH_HID_SetReport()

Description

Sends a report to a HID device. This essentially means sending data to the device.

Prototype

```
USBH_STATUS USBH_HID_SetReport(
            USBH_HID_HANDLE       hDevice,
            const U8            * pBuffer,
            U32                   BufferSize,
            USBH_HID_USER_FUNC  * pfFunc,
            void                * pContext);
```

Parameters

Parameter	Description
hDevice	Handle to an opened device.
pBuffer	Pointer to a caller allocated buffer. IN: data to write OUT: ---
BufferSize	Number of bytes to write.
pfFunc	Callback function of type USBH_HID_USER_FUNC invoked when the send operation finishes. It can be the NULL pointer. For further information see the "Additional information" section below.
pContext	Application specific pointer. The pointer is not dereferenced by the emUSB-Host. It is passed to the pfFunc callback function. Any value the application chooses is permitted, including NULL.

Table 6.14: USBH_HID_GetReport() parameter list

Return value

== USBH_STATUS_SUCCESS:	Read report sent
== USBH_STATUS_INVALID_PARAM:	An invalid handle was passed to the function
== USBH_STATUS_PENDING:	Request was submitted and application is informed via callback

Any other value means error.

Additional information

This function behaves differently whether the pfFunc points to a callback function or is a NULL pointer.

The write operation is asynchronous if pfFunc is != NULL. The function does not block until data is sent, instead it returns immediately. The emUSB-Host invokes the pfFunc callback function, in the context of the USBH_Task() routine, when the write operation ends. The written data is fetched by emUSB-Host directly from the pReport array so ensure the memory location pReport points to is valid until the callback is invoked.

If the pfFunc is NULL the write operation is synchronous. This means that the function blocks until all the data is sent to device.

You can stop the write operation at any time by calling the USBH_HID_CancelIo() function.

6.3 Data Structures

This chapter describes the used structures defined in the header file `USBH.h`.

Structure	Description
USBH_HID_DEVICE_INFO	Describes a HID device.
USBH_HID_KEYBOARD_DATA	Contains keyboard state information.
USBH_HID_MOUSE_DATA	Contains mouse state information.
USBH_HID_REPORT_INFO	Describes a HID report.

Table 6.15: emUSB-Host HID data structure overview

6.3.1 USBH_HID_DEVICE_INFO

Definition

```
typedef struct {
        U16  InputReportSize;
        U16  OutputReportSize;
        U16  ProductId;
        U16  VendorId;
        char acName[7];
} USBH_HID_DEVICE_INFO;
```

Description

Describes a HID device.

Members

Member	Description
InputReportSize	Size of the Input Report type in bytes.
OutputReportSize	Size of the Output Report type in bytes.
ProductId	USB Product ID.
VendorId	USB Vendor ID.
acName	Internal name of the HID device. (e.g. “hid001“)

Table 6.16: USBH_HID_DEVICE_INFO parameter list

6.3.2 USBH_HID_KEYBOARD_DATA

Definition

```
typedef struct {
        unsigned Code;
        int      Value;
} USBH_HID_KEYBOARD_DATA;
```

Description

Contains keyboard state information.

Members

Member	Description
Code	Contains the keycode.
Value	Keyboard state info. Refer to sample code for more information.

Table 6.17: USBH_HID_KEYBOARD_DATA parameter list

6.3.3 USBH_HID_MOUSE_DATA

Definition

```
typedef struct {
        int xChange;
        int yChange;
        int WheelChange;
        int ButtonState;
} USBH_HID_MOUSE_DATA;
```

Description

Contains mouse state information.

Members

Member	Description
xChange	Change of x-position since last event.
yChange	Change of y-position since last event.
WheelChange	Change of wheel-position since last event (if wheel is present).
ButtonState	Each bit corresponds to one button on the mouse. If the bit is set, the corresponding button is pressed. Typically, bit 0 corresponds to the left mouse button bit 1 corresponds to the right mouse button bit 2 corresponds to the middle mouse button.

Table 6.18: USBH_HID_MOUSE_DATA parameter list

6.3.4 USBH_HID_REPORT_INFO

Definition

```
typedef struct {
        USBH_HID_REPORT_TYPE ReportType;
        U32                  ReportId;
        U32                  ReportSize;
} USBH_HID_REPORT_INFO;
```

Description

Describes a HID report.

Members

Member	Description
ReportType	Type of report. It can be one of the following constants: • USBH_HID_INPUT_REPORT • USBH_HID_OUTPUT_REPORT • USBH_HID_FEATURE_REPORT
ReportId	Id of report.
ReportSize	Size of report in bytes.

Table 6.19: USBH_HID_REPORT_INFO parameter list

6.4 Function Types

This chapter describes the used structures defined in the header file USBH.h.

Type	Description
USBH_HID_ON_KEYBOARD_FUNC	This callback function is called when a key is pressed or released.
USBH_HID_ON_MOUSE_FUNC	This callback function is called when the mouse is moved or a button is pressed or released.
USBH_HID_USER_FUNC	Callback function is invoked when a report read/write operation finishes.

Table 6.20: emUSB-Host HID function type overview

6.4.1 USBH_HID_ON_KEYBOARD_FUNC

Definition

```
typedef void USBH_HID_ON_KEYBOARD_FUNC(USBH_HID_KEYBOARD_DATA * pKeyData);
```

Description

This callback function is called when a key is pressed or released.

Parameter

Parameter	Description
pKeyData	Points to the structure containing status information.

Table 6.21: USBH_HID_ON_KEYBOARD_FUNC() parameter list

6.4.2 USBH_HID_ON_MOUSE_FUNC

Definition

```
typedef void USBH_HID_ON_MOUSE_FUNC(USBH_HID_MOUSE_DATA * pMouseData);
```

Description

This callback function is called when the mouse is moved or a button is pressed or released.

Parameter

Parameter	Description
pMouseData	Points to the structure containing status information.

Table 6.22: USBH_HID_ON_MOUSE_FUNC() parameter list

6.4.3 USBH_HID_USER_FUNC

Definition

```
typedef void USBH_HID_USER_FUNC(void * pContext);
```

Description

This callback function is invoked when a report read/write operation finishes.

Parameter

Parameter	Description
`pContext`	Pointer passed as `pContext` parameter of `USBH_HID_GetReport()` and `USBH_HID_SetReport()` functions.

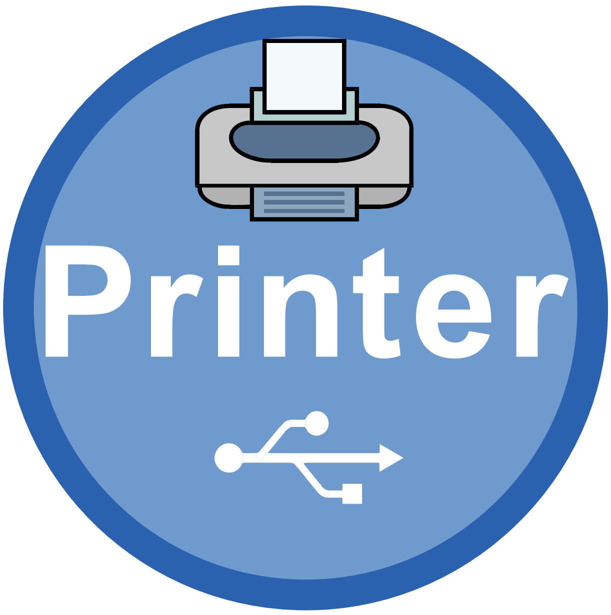
Table 6.23: USBH_HID_USER_FUNC() parameter list

Chapter 7

Printer Class (Add-On)

This chapter describes the emUSB-Host printer class software component and how to use it.

The printer class is an optional extenstion to emUSB-Host.



7.1 Introduction

The printer class software component of emUSB-Host allows the communication to USB printing devices. It implements the USB printer class protocol specified by the USB Implementers Forum.

This chapter describes the architecture, the features and the programming interface of this software component. To improve the readability of application code, all the functions and data types of this API are prefixed with the “USBH_PRINTER_” text.

In the following text the word “printer” is used to refer to any USB device that produces a hard copy of data sent to it.

7.1.1 Overview

A printer connected to the emUSB-Host is automatically configured and added to an internal list. The application receives a notification each time a printer is added or removed over a callback. In order to communicate to a printer the application should open a handle to it. The printers are identified by an index. The first connected printer gets assigned the index 0, the second index 1, and so on. You can use this index to identify a printer in a call to `USBH_PRINTER_OpenByIndex()` function.

7.1.2 Features

The following features are provided:

- Handling of multiple printers at the same time.
- Notifications about printer connection status.
- Ability to query the printer operating status and its device ID.

7.1.3 Example code

An example application which uses the API is provided in the OS_USBH_Printer.c file of your shipment. This example displays information about the printer and its connection status in the I/O terminal of the debugger. In addition the text “Hello World“ is printed out at the top of the current page when the first printer connects.

7.2 API Functions

This chapter describes the emUSB-Host printer API functions. These functions are defined in the header file USBH.h.

Function	Description
USBH_PRINTER_Close()	Closes a printer handle.
USBH_PRINTER_ConfigureTimeout()	Sets the timeout for the read an write operations.
USBH_PRINTER_ExecSoftReset()	Flushes all send and receive buffers.
USBH_PRINTER_Exit()	Releases all resources, closes all driver instances and unregisters all notification functions.
USBH_PRINTER_GetDeviceId()	Asks the USB printer to send the IEEE.1284 Id string.
USBH_PRINTER_GetNumDevices()	Returns the number of available printers.
USBH_PRINTER_GetPortStatus()	Returns the status of a printer.
USBH_PRINTER_Init()	Initializes the printer class driver.
USBH_PRINTER_Open()	Opens a handle to a printer using its name.
USBH_PRINTER_OpenByIndex()	Opens a handle to a printer using its index.
USBH_PRINTER_Read()	Receives data from a printer.
USBH_PRINTER_RegisterNotification()	Registers a notification for the printer connect/disconnect events.
USBH_PRINTER_Write()	Sends data to a printer.

Table 7.1: emUSB-Host printer class API function overview

7.2.1 USBH_PRINTER_Close()

Description

Closes a handle to an opened printer.

Prototype

```
USBH_STATUS USBH_PRINTER_Close(USBH_PRINTER_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Table 7.2: USBH_PRINTER_Close() parameter list

Return Value

`== USBH_STATUS_SUCCESS:`	Handle closed
`== USBH_STATUS_ERROR:`	Invalid handle

Additional Information

The function does not need to be called after the printer device was removed, since emUSB-Host handles removing the handle and freeing all resources.

7.2.2 USBH_PRINTER_ConfigureTimeout()

Description

Sets the timeout for the read and write operations.

Prototype

```
void USBH_PRINTER_ConfigureTimeout(U32 Timeout);
```

Parameter

Parameter	Description
Timeout	Operation timeout in milliseconds.

Table 7.3: USBH_PRINTER_ConfigureTimeout() parameter list

7.2.3 USBH_PRINTER_ExecSoftReset()

Description

Flushes all send and receive buffers.

Prototype

```
USBH_STATUS USBH_PRINTER_ExecSoftReset(
            USBH_PRINTER_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	Handle to the opened printer.

Table 7.4: USBH_PRINTER_ExecSoftReset() parameter list

Return Value

`== USBH_STATUS_SUCCESS:`	Reset executed
`== USBH_STATUS_ERROR:`	An error occurred

7.2.4 USBH_PRINTER_Exit()

Description

Releases all resources, closes all driver instances and unregisters all notification functions.

Prototype

```
void USBH_PRINTER_Exit();
```

Additional information

Has to be called if the application is closed before the USB bus driver is closed.

7.2.5 USBH_PRINTER_GetDeviceId()

Description

Asks the USB printer to send the IEEE.1284 Id string.

Prototype

```
USBH_STATUS USBH_PRINTER_GetDeviceId(
            USBH_PRINTER_HANDLE   hDevice,
            U8                  * pBuffer,
            unsigned              BufferSize);
```

Parameter

Parameter	Description
hDevice	Handle to the opened printer.
pBuffer	Pointer to a caller allocated buffer. IN: --- OUT: read device Id
BufferSize	Number of bytes allocated for the read buffer.

Table 7.5: USBH_PRINTER_GetDeviceId() parameter list

Return Value

```
== USBH_STATUS_SUCCESS:        Device Id read
== USBH_STATUS_ERROR:          An error occurred
```


7.2.6 USBH_PRINTER_GetNumDevices()

Description

Returns the number of available printers.

Prototype

```
int USBH_PRINTER_GetNumDevices();
```

Return Value

Number of available printers.

7.2.7 USBH_PRINTER_GetPortStatus()

Description

Returns the status of printer.

Prototype

```
USBH_STATUS USBH_PRINTER_GetPortStatus(
            USBH_PRINTER_HANDLE   hDevice,
            U8                  * pStatus);
```

Parameter

Parameter	Description
hDevice	Handle to the opened printer.
pStatus	Pointer to a caller allocated variable. IN: --- OUT: status of printer.

Table 7.6: USBH_PRINTER_GetPortStatus() parameter list

Return Value

== USBH_STATUS_SUCCESS:	Port status read.
== USBH_STATUS_ERROR:	An error occurred.

Additional information

The returned status is to be interpreted as follows:

Bit(s)	Field	Description
7 .. 6	Reserved	Reserved for future use; device shall return these bits reset to zero.
5	Paper Empty	1 = Paper Empty, 0 = Paper Not Empty
4	Select	1 = Selected, 0 = Not Selected
3	Not Error	1 = No Error, 0 = Error
2 .. 0	Reserved	Reserved for future use; device shall return these bits reset to zero.

7.2.8 USBH_PRINTER_Init()

Description

Initializes the USBH printer class library.

Prototype

```
USBH_STATUS USBH_PRINTER_Init();
```

Return value

== USBH_STATUS_SUCCESS:	Printer component initialized
== USBH_STATUS_ERROR:	An error occurred

Additional information

Performs basic initialization of the library. Has to be called before any other library function is called. It can be called again to reinitialize the library. In this case all internal states, such as added devices or handles, are lost.

7.2.9 USBH_PRINTER_Open()

Description

Opens a handle to a printer. The printer is identified by its name.

Prototype

```
USBH_PRINTER_HANDLE USBH_PRINTER_Open(const char * sName);
```

Parameters

Parameter	Description
sName	Name of the printer to open. It has the form prt*nnn* where *nnn* is 000 for the device with index 0, 001 for the device with index 1 and so on.

Table 7.7: USBH_PRINTER_Open() parameter list

Return Value

`!= 0`: Handle to a printer
`== 0`: Device not available

Additional Information

It is recommended to use `USBH_PRINTER_OpenByIndex()`.

7.2.10 USBH_PRINTER_OpenByIndex()

Description

Opens a handle to a printer. The printer is identified by its index.

Prototype

```
USBH_PRINTER_HANDLE USBH_PRINTER_OpenByIndex(U16 Index);
```

Parameters

Parameter	Description
Index	Index of the printer to open. The first printer has the index 0, the second index 1 and so on.

Table 7.8: USBH_PRINTER_Open() parameter list

Return Value

`!= 0`: Handle to a printer
`== 0`: Device not available

Additional Information

emUSB-Host assigns the smallest available index to each connected printer. The index remains the same as long as the printer is connected.

7.2.11 USBH_PRINTER_Read()

Description

Receives data from a printer.

Prototype

```
USBH_STATUS USBH_PRINTER_Read(
            USBH_PRINTER_HANDLE   hDevice,
            U8                  * pBuffer,
            unsigned              BufferSize);
```

Parameter

Parameter	Description
hDevice	Handle to the opened printer.
pBuffer	Pointer to a caller allocated buffer. IN: --- OUT: data received from printer
BufferSize	Size of the receive buffer in bytes.

Table 7.9: USBH_PRINTER_Read() parameter list

Return Value

== USBH_STATUS_SUCCESS:	Data received
== USBH_STATUS_ERROR:	An error occurred

Additional Information

Not all printers support read operation. For the normal usage of a printer, reading from the printer is normally not required. Some printers do not even provide an IN-Endpoint for read operations.

Typically a read operation can be used to feedback status information from the printer to the host. This type of feedback requires usually a command to be sent to the printer first. Which type of information can be read from the printer depends very much on the model.

7.2.12 USBH_PRINTER_RegisterNotification()

Description

Registers a notification for the printer connect/disconnect events.

Prototype

```
void USBH_PRINTER_RegisterNotification(
     USBH_NOTIFICATION_FUNC * pfNotification,
     void                   * pContext);
```

Parameter

Parameter	Description
pfNotification	Pointer to a function the library should call when a printer is connected or disconnected.
pContext	Pointer to a user context that should be passed to the callback function.

Table 7.10: USBH_PRINTER_RegisterNotification() parameter list

Additional Information

You can register only one notification function for all printers. To unregister, call this function with the pfNotification parameter set to NULL.

7.2.13 USBH_PRINTER_Write()

Description

Sends data to a printer.

Prototype

```
USBH_STATUS USBH_PRINTER_Write(
            USBH_PRINTER_HANDLE   hDevice,
            const U8            * pBuffer,
            unsigned              BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened printer.
pBuffer	Pointer to a caller allocated buffer. IN: --- OUT: data to send to printer.
BufferSize	Number of bytes to send.

Table 7.11: USBH_PRINTER_Write() parameter list

Return Value

== USBH_STATUS_SUCCESS:	Data sent
== USBH_STATUS_ERROR:	An error occurred

Additional Information

This functions does not alter the data it sends to printer. Data in ASCII form is typically printed out correctly by the majority of printers. For complex graphics the data passed to this function must be properly formatted according to the protocol the printer understands like Hewlett Packard PLC, IEEE 1284.1, Adode Postscript or Microsoft Windows Printing System (WPS).

Chapter 8

Mass Storage Device (MSD) class

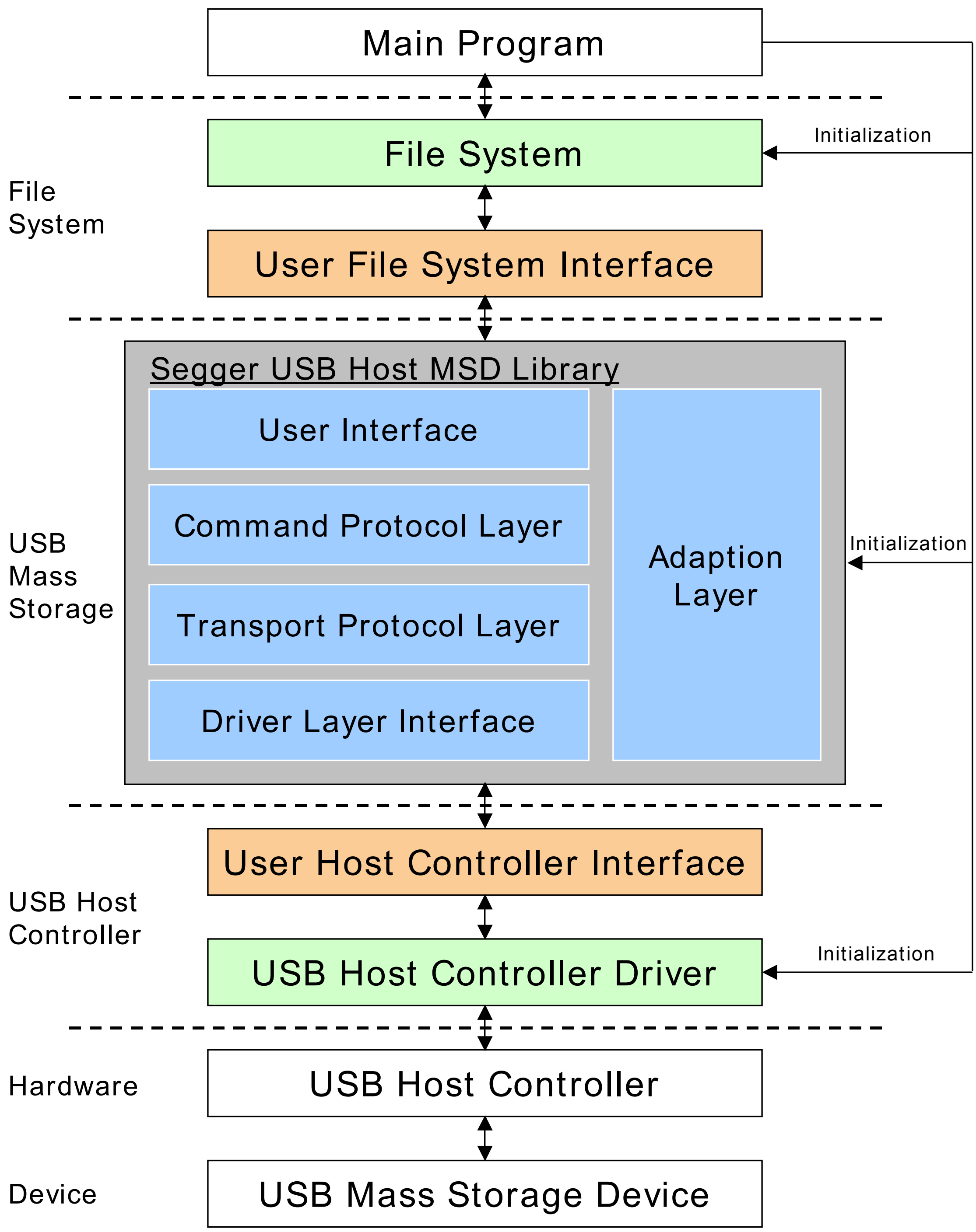
This chapter describes the emUSB-Host Mass storage device class driver and its usage.
The MSD class is part of the Core package. The MSD class code is linked in only if registered by the application program.

8.1 Introduction

The emUSB-Host MSD class software allows accessing USB Mass Storage Devices. It implements the USB Mass Storage Device class protocols specified by the USB Implementers Forum. The entire API of this class driver is prefixed "USBH_MSD_". This chapter describes the architecture, the features and the programming interface of the code.

8.1.1 Overview

A mass storage device connected to the emUSB-Host is added to the file system as device. All operations on the device, such as formatting, reading / writing of files and directories are performed through the API of the file system. With emFile, the device name of the first MSD is "msd:0:".
The structure of MSD component is shown in the following diagram:



8.1.2 Features

The following features are provided:

- The command block specification and protocol implementation used by the connected device will be automatically detected.
- It is independent of the file system. An interface to emFile is provided.

8.1.3 Restrictions

The following restrictions relate to the emUSB-Host library:

- The library supports only USB flash drives. Therefore not all protocol commands are implemented.

8.1.4 Requirements

To use the MSD class driver to perform file and directory operations, a file system (typically emFile) is required.

8.1.5 Example code

Example code which is provided in the file OS_USBH_MSD.c.

The example shows the capacity of the connected device, shows files in the root directory and creates and writes to a file.

8.1.6 Supported Protocols

The following table contains an overview about the implemented command protocols.

Command block specification	Implementation	Related documents
SCSI transparent command set	All necessary commands for accessing flash devices.	Mass Storage Class Specification Overview Revision 1.2., SCSI-2 Specification September 1993 Rev.10 (X3T9.2 Project 275D)
SFF-8070i	All necessary commands for accessing flash devices.	SFF-8070i Specification for ATAPI Removable Rewritable Media Devices (SFF Committee: document SFF-8070 Rev 1.3)

The following table contains an overview about the implemented transport protocols.

Protocol implementation	Implementation	Related documents
Bulk-Only transport	All commands implemented.	Universal Serial Bus Mass Storage Class Bulk-Only Transport Rev.1.0.

8.2 API Functions

This chapter describes the emUSB-Host MSD API functions. These functions are defined in the header file USBH.h.

Function	Description
USBH_MSD_Exit()	Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.
USBH_MSD_GetStatus()	Checks the state of a device unit.
USBH_MSD_GetUnitInfo()	Returns basic information about the logical unit (LUN).
USBH_MSD_Init()	Initializes the USBH MSD library.
USBH_MSD_ReadSectors()	Reads sectors from a USB Mass Storage device.
USBH_MSD_WriteSectors()	Writes sectors to a USB Mass Storage device.

Table 8.1: emUSB-Host MSD API function overview

8.2.1 USBH_MSD_Exit()

Description

Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.

Prototype

```
void USBH_MSD_Exit(void);
```

Additional information

Has to be called if the application is closed before the USB bus driver is closed.

8.2.2 USBH_MSD_GetStatus()

Description

Checks the state of a device unit.

Prototype

```
USBH_STATUS USBH_MSD_GetStatus(U8 UnitId);
```

Parameter

Parameter	Description
UnitId	0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.

Table 8.2: USBH_MSD_GetStatus() parameter list

Return Value

If the device is working, USBH_STATUS_SUCCESS is returned. If the device does not work correctly or is disconnected the function returns USBH_STATUS_ERROR.

8.2.3 USBH_MSD_GetUnitInfo()

Description

Returns basic information about the logical unit (LUN).

Prototype

```
USBH_STATUS USBH_MSD_GetUnitInfo(
            U8                      UnitId,
            USBH_MSD_UNIT_INFO   * pInfo);
```

Parameters

Parameter	Description
UnitId	0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.
pInfo	Pointer to a caller provided storage buffer. It receives the information about the LUN in case of success.

Table 8.3: USBH_MSD_GetUnitInfo() parameter list

Return Value

Returns USBH_STATUS_SUCCESS in case of success. If the device is not a USB Mass Storage device, USBH_STATUS_ERROR will be returned. USBH_STATUS_TIMEOUT is returned if the function call timed out.

8.2.4 USBH_MSD_Init()

Description

Initializes the USBH MSD library.

Prototype

```
int USBH_MSD_Init(
    USBH_MSD_LUN_NOTIFICATION_FUNC  * pfLunNotification,
    void                            * pContext);
```

Parameters

Parameter	Description
pfLunNotification	Pointer to a function that shall be called when a new device notification is received. The function is called when a device is attached and ready or when it is removed.
pContext	Pointer to a context that should be passed when the `pfLunNotification` is called.

Table 8.4: USBH_MSD_Init() parameter list

Return value

`== 1:` On success
`== 0:` In case of an error

Additional information

Performs basic initialization of the library. Has to be called before any other library function is called. It can be called again to reinitialize the library. In this case all internal states like added devices or handles are lost.

Example:

```
/*********************************************************************
*
*       _cbOnAddRemoveDevice
*
*  Function description
*    Callback, called when a device is added or removed.
*    Call in the context of the USBH_Task.
*    The functionality in this routine should not block
*/

static void _cbOnAddRemoveDevice(
       void             * pContext,
       U8                 DevIndex,
       USBH_MSD_EVENT   Event) {
  switch (Event) {
  case USBH_MSD_EVENT_ADD:
    printf("\n**** Device added\n");
    break;
  case USBH_MSD_EVENT_REMOVE:
    printf("\n**** Device removed\n");
    break;
  default:;    // Should never happen
  }
}

//
// Init MSD, after call to FS_Init(). See example code in OS_USBH_MSD.c
//
{
  USBH_MSD_Init(_cbOnAddRemoveDevice, NULL);
}
```


8.2.5 USBH_MSD_ReadSectors()

Description

Reads sectors from a USB Mass Storage device.

Prototype

```
USBH_STATUS USBH_MSD_ReadSectors(
            U8     UnitId,
            U32    SectorIndex,
            U32    NumSectors,
            U8   * pBuffer);
```

Parameters

Parameter	Description
UnitId	0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.
SectorIndex	0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, and so on.
NumSectors	Determines the number of sectors to read.
pBuffer	Pointer to a caller allocated buffer. IN: --- OUT: data of read sectors.

Table 8.5: USBH_MSD_ReadSectors() parameter list

Return Value

Returns `USBH_STATUS_SUCCESS` if the sectors have been successfully read from the device and copied to the Buffer. If reading from the specified device fails, the function returns `USBH_STATUS_READ` to indicate the error.

8.2.6 USBH_MSD_WriteSectors()

Description

Writes sectors to a USB Mass Storage device.

Prototype

```
USBH_STATUS USBH_MSD_WriteSectors(
            U8          UnitId,
            U32         SectorIndex,
            U32         NumSectors,
            const U8  * pBuffer);
```

Parameters

Parameter	Description
UnitId	0-based Unit Id. The first unit in the system has UnitId of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIds at run time.
SectorIndex	0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, and so on.
NumSectors	Determines the number of sectors to write.
pBuffer	Pointer to a caller allocated buffer. IN: data to write OUT: ---

Table 8.6: USBH_MSD_WriteSectors() parameter list

Return Value

Returns USBH_STATUS_SUCCESS if the sectors have been successfully copied from the Buffer and written to the device. If writing to the specified device fails the function returns USBH_STATUS_WRITE to indicate the error. The function returns USBH_STATUS_WRITE_PROTECT if the medium is write protected.

8.3 Data Structures

This chapter describes the used structures defined in the header file USBH.h.

Structure	Description
USBH_MSD_UNIT_INFO	Contains logical unit information.

Table 8.7: emUSB-Host MSD data structure overview

8.3.1 USBH_MSD_UNIT_INFO

Definition

```
typedef struct USB_MSD_UNIT_INFO {
               U32  TotalSectors;
               U16  BytesPerSector;
               int  WriteProtectFlag;
               U16  VendorId;
               U16  ProductId;
               char acVendorName[9];
               char acProductName[17];
               char acRevision[5];
} USBH_MSD_UNIT_INFO;
```

Description

Contains logical unit information.

Members

Member	Description
TotalSectors	Contains the number of total sectors available on the LUN.
BytesPerSector	Contains the number of bytes per sector.
WriteProtectFlag	Nonzero if the device is write protected.
VendorId	USB Vendor ID.
ProductId	USB Product ID.
acVendorName	Vendor identification string.
acProductName	Product identification string.
acRevision	Revision string.

Table 8.8: USBH_MSD_UNIT_INFO parameter list

8.4 Function Types

This chapter describes the used structures defined in the header file USBH_MSD.h.

Type	Description
USBH_MSD_LUN_NOTIFICATION_FUNC	Type of callback set in USBH_MSD_Init().

Table 8.9: emUSB-Host MSD function type overview

8.4.1 USBH_MSD_LUN_NOTIFICATION_FUNC

Definition

```
typedef void USB_MSD_LUN_NOTIFICATION_FUNC(
             void            * pContext;
             U8                DevIndex;
             USBH_MSD_EVENT    Event);
```

Description

This callback function is called when a logical unit is either added or removed. To get detailed information USBH_MSD_GetStatus() has to be called. The LUN indexes must be used to get access to a specified unit of the device.

Parameters

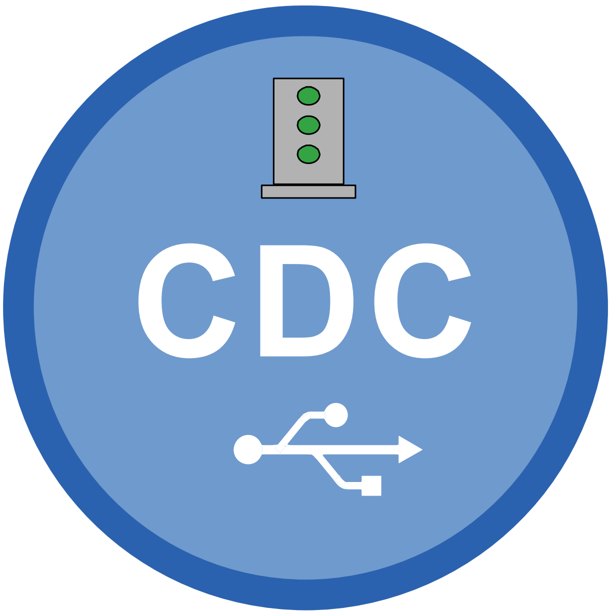
Parameter	Description
pContext	Pointer to a context that was set by the user when the USBH_MSD_Init() was called.
DevIndex	Zero based index of the device that was attached or removed. First device has index 0, second one has index 1, etc.
Event	Gives information about the event that has occurred. The following events are currently available: • USBH_MSD_EVENT_ADD_LUN A device was attached. • USBH_MSD_EVENT_REMOVE_LUN A device was removed.

Table 8.10: USBH_MSD_LUN_NOTIFICATION_FUNC() parameter list

Chapter 9

CDC Device Driver (Add-On)

This chapter describes the optional emUSB-Host add-on "CDC device driver". It allows communication with a CDC USB device.



9.1 Introduction

The CDC driver software component of emUSB-Host allows communication with CDC devices. The Communication Device Class (CDC) is an abstract USB class protocol defined by the USB Implementers Forum. The protocol allows emulation of serial communication via USB.

This chapter provides an explanation of the functions available to application developers via the CDC driver software. All the functions and data types of this add-on are prefixed with the "USBH_CDC_" text.

9.1.1 Overview

A CDC device connected to the emUSB-Host is automatically configured and added to an internal list. If the CDC driver has been registered, it is notified via a callback when a CDC device has been added or removed. The driver then can notify the application program, when a callback function has been registered via `USBH_CDC_RegisterNotification()`. In order to communicate with such a device, the application has to call the `USBH_CDC_Open()`, passing the device index. CDC devices are identified by an index. The first connected device gets assigned the index 0, the second index 1, and so on.

9.1.2 Features

The following features are provided:

- Compatibility with different CDC devices.
- Ability to send and receive data.
- Ability to set various parameters, such as baudrate, number of stop bits, parity.
- Handling of multiple CDC devices at the same time.
- Notifications about CDC connection status.
- Ability to query the CDC line and modem status.

9.1.3 Example code

An example application which uses the API is provided in the OS_USBH_CDC.c file. This example displays information about the CDC device in the I/O terminal of the debugger. In addition the application then starts a simple echo server, sending back the received data.

9.2 API Functions

This chapter describes the emUSB-Host CDC driver API functions. These functions are defined in the header file `USBH_CDC.h`.

Function	Description
USBH_CDC_Init()	Initializes the CDC device driver.
USBH_CDC_Exit()	Deinitialize the CDC device driver.
USBH_CDC_RegisterNotification()	Sets a callback in order to be notified when a device is added or removed.
USBH_CDC_ConfigureDefaultTimeout()	Sets the default read and write timeout that shall be used when a new device is connected.
USBH_CDC_Open()	Opens a device given by an index.
USBH_CDC_Close()	Closes a handle to an opened device.
USBH_CDC_AllowShortRead()	The configuration function allows to let the read function to return as soon as data are available.
USBH_CDC_GetDeviceInfo()	Retrieves the information about the CDC device.
USBH_CDC_SetTimeouts()	Sets up the default timeouts the host waits until the data transfer is aborted
USBH_CDC_Read()	Reads data from the CDC device.
USBH_CDC_Write()	Writes data to the CDC device.
USBH_CDC_ReadAsync()	Triggers a read transfer, receive result via callback.
USBH_CDC_WriteAsync()	Triggers a write transfer, receive result via callback.
USBH_CDC_SetCommParas()	Setups the serial communication with the given characteristics.
USBH_CDC_SetDtr()	Sets the Data Terminal Ready (DTR) control signal.
USBH_CDC_ClrDtr()	Clears the Data Terminal Ready (DTR) control signal.
USBH_CDC_SetRts()	Sets the Request To Send (RTS) control signal.
USBH_CDC_ClrRts()	Clears the Request To Send (RTS) control signal.
USBH_CDC_GetQueueStatus()	Gets the number of bytes in the receive queue.
USBH_CDC_SetBreak()	Sets the BREAK condition for the device for a specific amount of time.
USBH_CDC_SetBreakOn()	Sets the BREAK condition for the device.
USBH_CDC_SetBreakOff()	Resets the BREAK condition for the device.
USBH_CDC_GetSerialState()	Gets the modem status and line status from the device.
USBH_CDC_AddDevice()	Allows adding non-standard CDC devices to the CDC modules.
USBH_CDC_RemoveDevice()	Removes devices which were added by USBH_CDC_AddDevice()
USBH_CDC_SetConfigFlags()	Sets global configuration flags.

Table 9.1: emUSB-Host CDC device driver API function overview

9.2.1 USBH_CDC_Init()

Description

Initializes and registers the CDC device driver to emUSB-Host.

Prototype

```
USBH_BOOL USBH_CDC_Init(void);
```

Return Value

`== 1:` Success
`== 0:` Could not register CDC device driver

9.2.2 USBH_CDC_Exit()

Description

Unregisters and deinitializes the CDC device driver from emUSB-Host.

Prototype

```
void USBH_CDC_Exit(void);
```

Additional information

This function will release resources that were used by this device driver. It has to be called if the application is closed. This has to be called before `USBH_Exit()` is called. No more functions of this module may be called after calling `USBH_CDC_Exit()`. The only exception is `USBH_CDC_Init()`, which would in turn re-init the module and allow further calls.

9.2.3 USBH_CDC_RegisterNotification()

Description

Sets a callback in order to be notified when a device is added or removed.

Prototype

```
void USBH_CDC_RegisterNotification(
     USBH_NOTIFICATION_FUNC * pfNotification,
     void                   * pContext);
```

Parameter

Parameter	Description
pfNotification	[IN] - Pointer to a function the stack should call when a device is connected or disconnected.
pContext	[IN] - Pointer to a user context that should be passed to the callback function.

Table 9.2: USBH_CDC_RegisterNotification() parameter list

Additional Information

Only one notification function can be set for all devices. To unregister, call this function with the pfNotification parameter set to NULL.

Example

```
/*********************************************************************
*
*       _cbOnAddRemoveDevice
*
*   Function description
*     Callback, called when a device is added or removed.
*     Call in the context of the USBH_Task.
*     The functionality in this routine should not block
*/
static void _cbOnAddRemoveDevice(void * pContext, U8 DevIndex, USBH_DEVICE_EVENT
Event) {
  pContext = pContext;  // avoid "never referenced" warning
  switch (Event) {
  case USBH_DEVICE_EVENT_ADD:
    printf("\n**** Device added\n");
    _DevIndex = DevIndex;
    _DevIsReady = 1;
    break;
  case USBH_DEVICE_EVENT_REMOVE:
    printf("\n**** Device removed\n");
    _DevIsReady = 0;
    _DevIndex   = -1;
    _Removed    = 1;
    break;
  default:;   // Should never happen
  }
}
```



```
/*********************************************************************
*
*       CDC_Task
*
*  Function description
*    This task shall handle CDC devices. It initialize the CDC driver
*    and sets a  notification callback in order to be informed about adding
*    removing of CDC devices.
*/
void CDC_Task(void) {
  USBH_CDC_Init();
  USBH_CDC_RegisterNotification(_cbOnAddRemoveDevice, NULL);
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay(100);
    if (_DevIsReady) {
      _OnDevReady();
    }
  }
}
```


9.2.4 USBH_CDC_ConfigureDefaultTimeout()

Description

Sets the default read and write timeout that shall be used when a new device is connected.

Prototype

```
void USBH_CDC_ConfigureDefaultTimeout(U32 ReadTimeout,
                                     U32 WriteTimeout);
```

Parameter

Parameter	Description
ReadTimeout	[IN] - Default read timeout given in ms.
WriteTimeout	[IN] - Default write timeout given in ms.

Table 9.3: USBH_CDC_ConfigureDefaultTimeout() parameter list

Additional information

The function shall be called after USBH_CDC_Init() has been called, otherwise the behavior is undefined.

Example

```
void CDC_Task(void) {
  USBH_CDC_Init();
  USBH_CDC_ConfigureDefaultTimeout(50, 50); // Configure default timeout for read and
                                            //  write to 50ms.
[...]
```


9.2.5 USBH_CDC_Open()

Description

Opens a device given by an index.

Prototype

```
USBH_CDC_HANDLE USBH_CDC_Open(unsigned Index);
```

Parameter

Parameter	Description
Index	[IN] - Index of the device that shall be opened. In general this means: the first connected device is 0, second device is 1 etc.

Table 9.4: USBH_CDC_Open() parameter list

Return Value

`== 0`: Device could not be opened (Removed or not available)
`!= 0`: Handle to the device

Example

```
USBH_CDC_HANDLE hDevice;

hDevice = USBH_CDC_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.6 USBH_CDC_Close()

Description

Closes a handle to an opened device.

Prototype

```
USBH_STATUS USBH_CDC_Close(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the device which shall be closed.

Table 9.5: USBH_CDC_Close() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_CDC_HANDLE hDevice;
U32             NumBytesWritten;

hDevice = USBH_CDC_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  USBH_CDC_Write(hDevice, "Hello\n", 6, &NumBytesWritten);
  USBH_CDC_Close(hDevice);
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.7 USBH_CDC_AllowShortRead()

Description

The configuration function allows to let the read function to return as soon as data are available.

Prototype

```
USBH_STATUS USBH_CDC_AllowShortRead(USBH_CDC_HANDLE hDevice,
                                    U8               AllowShortRead);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
AllowShortRead	[IN] - Define whether short read mode shall be used or not. 1 - Enable short read mode 0 - Disable short read mode.

Table 9.6: USBH_CDC_AllowShortRead() parameter list

Return Value

`== USBH_STATUS_SUCCESS` Success
Any other value means error.

Additional information

USBH_CDC_AllowShortRead() sets the USBH_CDC_Read into a special mode - short read mode. When this mode is enabled, the function returns as soon as data was read from the device. This allows the application to read data where the number of bytes to read is undefined.
To disable this mode, AllowShortRead shall be 0.

Example

```
USBH_CDC_HANDLE      hDevice;
USBH_CDC_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;
U32                    NumBytesWritten;
U32                    NumBytes2Write = 6;

hDevice = USBH_CDC_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_CDC_Write(hDevice, "Hello\n", NumBytes2Write, &NumBytesWritten);
  if (Status == USBH_STATUS_SUCCESS) {
    printf("All bytes have been written!\n");
  } else {
    printf("Not all bytes (%d of %d) have been written, error code = 0x%x",
                                                         NumBytesWritten,
                                                         NumBytes2Write,
                                                         Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.8 USBH_CDC_GetDeviceInfo()

Description

Retrieves the information about the CDC device.

Prototype

```
USBH_STATUS USBH_CDC_GetDeviceInfo(USBH_CDC_HANDLE        hDevice,
                                   USBH_CDC_DEVICE_INFO * pDevInfo);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pDevInfo	[OUT] - Pointer to a USBH_SCDC_DEVICE_INFO structure to store information related to the device.

Table 9.7: USBH_CDC_GetDeviceInfo() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_CDC_HANDLE      hDevice;
USBH_CDC_DEVICE_INFO DeviceInfo;

hDevice = USBH_CDC_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  USBH_CDC_GetDeviceInfo(hDevice, &DeviceInfo);
  printf("Vendor  ID = 0x%4.4x\n"
         "Product ID = 0x%4.4x\n", DeviceInfo.VendorId,
                                   DeviceInfo.ProductId);
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.9 USBH_CDC_SetTimeouts()

Description

Sets up the timeouts for a specific device, referenced by the CDC handle, the host waits until the data transfer is aborted.

Prototype

```
USBH_STATUS USBH_CDC_SetTimeouts(USBH_CDC_HANDLE hDevice,
                                 U32             ReadTimeout,
                                 U32             WriteTimeout);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
ReadTimeout	[IN] - Read timeout given in ms.
WriteTimeout	[IN] - Write timeout given in ms.

Table 9.8: USBH_CDC_SetTimeouts() parameter list

Return Value

USBH_STATUS_SUCCESS: Success.
Any other value means error.

Example

```
USBH_CDC_HANDLE       hDevice;
USBH_CDC_DEVICE_INFO  DeviceInfo;
USBH_STATUS           Status;

hDevice = USBH_CDC_Open(0); // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
   Status = USBH_CDC_SetTimeouts(hDevice, 30, 30); // Set timeout for both to 
30ms.
  if (Status == USBH_STATUS_SUCCESS) {
    printf("Setting the timeout was successful!\n");
  } else {
    printf("Failed to set timeout, Error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.10 USBH_CDC_Read()

Description

Reads data from the CDC device.

Prototype

```
USBH_STATUS USBH_CDC_Read (USBH_CDC_HANDLE hDevice,
                           U8  *           pData,
                           U32             NumBytes,
                           U32 *           pNumBytesRead);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pData	[IN] - Pointer to the buffer that receives the data from the device.
NumBytes	[IN] - Number of bytes to be read from the device.
pNumBytesRead	[OUT] - Pointer to a variable of type U32 which receives the number of bytes read from the device.

Table 9.9: USBH_CDC_Read() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

USBH_CDC_Read() always returns the number of bytes read in pNumBytesRead.
This function does not return until NumBytes bytes have been read into the buffer or the timeout has been reached, unless short read mode is enabled (via USBH_CDC_AllowShortRead()). The short read mode allows USBH_CDC_Read() to return when either data has been read from the queue or as soon as some data has been read from the device.
The number of bytes in the receive queue can be determined by calling USBH_CDC_GetQueueStatus(), and passed to USBH_CDC_Read() as NumBytes so that the function reads the device and returns immediately.
When a read timeout value has been specified in a previous call to USBH_CDC_SetTimeouts(), USBH_CDC_Read() returns when the timer expires or NumBytes have been read, whichever occurs first. If the timeout occurred, USBH_CDC_Read() reads available data into the buffer and returns USBH_STATUS_TIMEOUT. If a timeout is not specified via USBH_CDC_SetTimeouts() the default timeout is used, which is 5000 ms.
An application should use the function return value and pNumBytesRead when processing the buffer. If the return value is USBH_STATUS_SUCCESS, and pNumBytesReturned is equal to NumBytes then USBH_CDC_Read() has completed normally.
If the return value is USBH_STATUS_TIMEOUT, pNumBytesRead may be less or even 0, in any case, pData will filled with pNumBytesRead.
Any other return value suggests an error in the parameters of the function, or a fatal error like a USB disconnect.

9.2.11 USBH_CDC_Write()

Description

Writes data to the CDC device. The function blocks until all data has been written or until the timeout has been reached. If a timeout is not specified via USBH_CDC_SetTimeouts() the default timeout is used, which is 5000 ms.

Prototype

```
USBH_STATUS USBH_CDC_Write(    USBH_CDC_HANDLE hDevice,
                               const U8 *      pData,
                               U32             NumBytes,
                                 U32 *            pNumBytesWritten);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pData	[IN] - Pointer to the buffer that contains the data to be written to the device.
NumBytes	[IN] - Number of bytes to write to the device.
pNumBytesWritten	[OUT] - Pointer to a variable of type U32 which receives the number of bytes written to the device.

Table 9.10: USBH_CDC_Write() parameter list

Return Value

== USBH_STATUS_SUCCESS: Success
Any other value means error.

Example

```
USBH_CDC_HANDLE      hDevice;
USBH_CDC_DEVICE_INFO DeviceInfo;
USBH_STATUS          Status;
U32                  NumBytes;

hDevice = USBH_CDC_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_CDC_Write(hDevice, "SEGGER", 7, &NumBytes);// Write SEGGER\0 over CDC
  if (Status == USBH_STATUS_SUCCESS) {
    printf("Write was successful!\n");
  } else {
    printf("Failed to write, Error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.12 USBH_CDC_ReadAsync()

Description

Triggers a read transfer to the CDC device. The result of the transfer is received through the user callback.

Prototype

```
USBH_STATUS USBH_CDC_ReadAsync(USBH_CDC_HANDLE hDevice,
                               U8 * pBuffer,
                               U32 BufferSize,
                               USBH_CDC_ON_COMPLETE_FUNC * pfOnComplete,
                               USBH_CDC_RW_CONTEXT * pRWContext);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pBuffer	[IN] - Pointer to the buffer that receives the data from the device.
BufferSize	[IN] - Size of the buffer in bytes.
pfOnComplete	[IN] - Pointer to a user function which will be called after the transfer has been completed.
pRWContext	[IN] - Pointer to a USBH_CDC_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function.

Table 9.11: USBH_CDC_Read() parameter list

Return Value

== USBH_STATUS_SUCCESS:	Success
== USBH_STATUS_PENDING:	Success, the data transfer is queued, the user callback will be called

Any other value means error.

Additional information

View the descriptions of USBH_CDC_ON_COMPLETE_FUNC and USBH_CDC_RW_CONTEXT for a better understanding of this function’s usage.

9.2.13 USBH_CDC_WriteAsync()

Description

Triggers a write transfer to the CDC device. The result of the transfer is received through the user callback.

Prototype

```
USBH_STATUS USBH_CDC_WriteAsync(USBH_CDC_HANDLE hDevice,
                                U8 * pBuffer,
                                U32 BufferSize,
                                USBH_CDC_ON_COMPLETE_FUNC * pfOnComplete,
                                USBH_CDC_RW_CONTEXT * pRWContext);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pBuffer	[IN] - Pointer to a buffer which holds the data.
BufferSize	[IN] - Size of the buffer in bytes.
pfOnComplete	[IN] - Pointer to a user function which will be called after the transfer has been completed.
pRWContext	[IN] - Pointer to a USBH_CDC_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function.

Table 9.12: USBH_CDC_Write() parameter list

Return Value

== USBH_STATUS_SUCCESS:	Success
== USBH_STATUS_PENDING:	Success, the data transfer is queued, the user callback will be called

Any other value means error.

Additional information

View the descriptions of USBH_CDC_ON_COMPLETE_FUNC and USBH_CDC_RW_CONTEXT for a better understanding of this function’s usage.

9.2.14 USBH_CDC_SetCommParas()

Description

Setups the serial communication with the given characteristics.

Prototype

```
USBH_STATUS USBH_CDC_SetCommParas(USBH_CDC_HANDLE hDevice,
                                  U32             Baudrate,
                                  U8              DataBits,
                                  U8              StopBits,
                                  U8              Parity);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Baudrate	[IN] - Baud rate to set.
DataBits	[IN] - Number of data bits, can be 5, 6, 7, 8 or 16.
StopBits	[IN] - Number of stop bits. Must be USBH_CDC_STOP_BITS_1 or USBH_CDC_STOP_BITS_2
Parity	[IN] - Parity - must be one of the following values: UBSH_CDC_PARITY_NONE, UBSH_CDC_PARITY_ODD, UBSH_CDC_PARITY_EVEN, UBSH_CDC_PARITY_MARK UBSH_CDC_PARITY SPACE

Table 9.13: USBH_CDC_SetBaudRate() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_CDC_HANDLE      hDevice;
USBH_CDC_DEVICE_INFO DeviceInfo;
USBH_STATUS          Status;

hDevice = USBH_CDC_Open(0); // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_CDC_SetCommParas(hDevice,
                                 115200,
                                 USBH_CDC_STOP_BITS_1,
                                 UBSH_CDC_PARITY_NONE);
  if (Status == USBH_STATUS_SUCCESS) {
    printf("USBH_CDC_SetCommParas was successful!\n");
  } else {
    printf("Could not set baudrate, error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


9.2.15 USBH_CDC_SetDtr()

Description

Sets the Data Terminal Ready (DTR) control signal.

Prototype

```
USBH_STATUS USBH_CDC_SetDtr(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.14: USBH_CDC_SetDtr() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.16 USBH_CDC_ClrDtr()

Description

Clears the Data Terminal Ready (DTR) control signal.

Prototype

```
USBH_STATUS USBH_CDC_ClrDtr(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.15: USBH_CDC_ClrDtr() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.17 USBH_CDC_SetRts()

Description

Sets the Request To Send (RTS) control signal.

Prototype

```
USBH_STATUS USBH_CDC_SetRts(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.16: USBH_CDC_SetRts() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.18 USBH_CDC_ClrRts()

Description

Clears the Request To Send (RTS) control signal.

Prototype

```
USBH_STATUS USBH_CDC_ClrRts(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.17: USBH_CDC_ClrRts() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.19 USBH_CDC_GetQueueStatus()

Description

Gets the number of bytes in the receive queue.

This function is useful when the amount of bytes being received is not known.
Due to the fact that the USB controller can not check how many bytes it is going to receive it has to read a least one packet during a read transfer. But the packet itself can hold from 1 and up to MAX_PACKET_SIZE bytes of data.

Prototype

```
USBH_STATUS USBH_CDC_GetQueueStatus(USBH_CDC_HANDLE hDevice,
                                    U32 *           pRxBytes);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pRxBytes	[OUT] - Pointer to a variable of type U32 which receives the number of bytes in the receive queue.

Table 9.18: USBH_CDC_GetQueueStatus() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
//
// Read only ONE byte to trigger the read transfer.
// This means that the remaining bytes are in the internal packet buffer!
//
USBH_CDC_Read(hDevice, acData, 1, &NumBytes);
if (NumBytes) {
  //
  // We do not know how big the packet was which we received from the device,
  // since we only read 1 byte from the packet.
  // Therefore we still might have some data in the internal buffer!
  // Using USBH_CDC_GetQueueStatus we can check how many bytes are still in the
  // internal buffer (if any) and read those as well.
  //
  USBH_CDC_GetQueueStatus(hDevice, &RxBytes);
  //
  // Read the remaining bytes.
  //
  if (RxBytes > 0) {
    USBH_CDC_Read(hDevice, &acData[1], RxBytes, &NumBytes);
  }
}
```


9.2.20 USBH_CDC_SetBreak()

Description

Sets the BREAK condition for the device for a limited time.

Prototype

```
USBH_STATUS USBH_CDC_SetBreak(USBH_CDC_HANDLE hDevice,
                              U16             Duration);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Duration	[IN] - Duration of the BREAK condition in ms.

Table 9.19: USBH_CDC_SetBreakOn() parameter list

Return Value

== USBH_STATUS_SUCCESS: Success
Any other value means error.

9.2.21 USBH_CDC_SetBreakOn()

Description

Sets the BREAK condition for the device to "on" permanently.

Prototype

```
USBH_STATUS USBH_CDC_SetBreakOn(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.20: USBH_CDC_SetBreakOn() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.22 USBH_CDC_SetBreakOff()

Description

Resets the BREAK condition for the device.

Prototype

```
USBH_STATUS USBH_CDC_SetBreakOff(USBH_CDC_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 9.21: USBH_CDC_SetBreakOff() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

9.2.23 USBH_CDC_GetSerialState()

Description

Gets the modem status and line status from the device. The least significant byte of the pSerialState value holds the modem status. The line status is held in the second least significant byte of the pSerialState value.

Prototype

```
USBH_STATUS USBH_CDC_GetSerialState (USBH_CDC_HANDLE        hDevice,
                                     USBH_CDC_SERIALSTATE * pSerialState);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pSerialState	[OUT] - Pointer to a variable of type USBH_CDC_SERIALSTATE which receives the serial status from the device.

Table 9.22: USBH_CDC_GetSerialState() parameter list

Return Value

`== USBH_STATUS_SUCCESS` Success
Any other value means error.

Additional information

The least significant byte of the pSerialState value holds the modem status.
The line status is held in the second least significant byte of the pSerialState value.

The status is bit-mapped as follows:

```
Data Carrier Detect (DCD) = 0x01,
Data Set Ready      (DSR) = 0x02,
Break Interrupt    (BI)  = 0x04,
Ring Indicator     (RI)  = 0x08,
Framing Error      (FE)  = 0x10,
Parity Error       (PE)  = 0x20,
Overrun Error      (OE)  = 0x40.
```


9.2.24 USBH_CDC_AddDevice()

Description

This function should not be used for CDC compliant devices!
This function allows the application to register a device with a non-standard interface layout as a CDC device. After registering the device the application will receive ADD and REMOVE notifications to the user callback which was set by USBH_CDC_RegisterNotification().

Prototype

```
USBH_STATUS USBH_CDC_AddDevice(USBH_INTERFACE_ID ACMInterfaceId,
                               USBH_INTERFACE_ID DataInterfaceId,
                                 unsigned Flags)
```

Parameter

Parameter	Description
ACMInterfaceId	[IN] - Numeric index of the CDC ACM interface.
DataInterfaceId	[IN] - Numeric index of the CDC Data interface.
Flags	[IN] - Reserved for future use. Write zero.

Table 9.23: USBH_CDC_AddDevice() parameter list

Return Value

== USBH_STATUS_SUCCESS Success
Any other value means error.

Additional information

The numeric interface IDs can be retrived by setting up a PnP notification via USBH_RegisterPnPNotification(). Please note that the PnP notification callback will be triggered for each interface, but you only have to add the device once.
Alternatively you can simply set the IDs if you know the interface layout.

9.2.25 USBH_CDC_RemoveDevice()

Description

Removes the non-standard CDC device which was added by USBH_CDC_AddDevice().

Prototype

```
USBH_STATUS USBH_CDC_RemoveDevice(USBH_INTERFACE_ID ACMInterfaceId,
                                   USBH_INTERFACE_ID DataInterfaceId)
```

Parameter

Parameter	Description
ACMInterfaceId	[IN] - Numeric index of the CDC ACM interface.
DataInterfa-ceId	[IN] - Numeric index of the CDC Data interface.

Table 9.24: USBH_CDC_RemoveDevice() parameter list

Return Value

`== USBH_STATUS_SUCCESS` Success
Any other value means error.

9.2.26 USBH_CDC_SetConfigFlags()

Description

Sets configuration flags for the CDC module.

Prototype

```
void USBH_CDC_SetConfigFlags(U32 Flags)
```

Parameter

Parameter	Description
Flags	[IN] - Bitwise OR of the available configuration flags.

Table 9.25: USBH_CDC_SetConfigFlags() parameter list

Additional information

The following flags are avaialble:

Flag	Description
USBH_CDC_IGNORE_INT_EP	[IN] - Ignore that an Interrupt EP is available (ACM). It can also be used where no Interrupt EP is available.

9.3 Data Structures

This chapter describes the used structures defined in the header file USBH_CDC.h.

Structure	Description
USBH_CDC_DEVICE_INFO	Contains information about a CDC compatible device.
USBH_CDC_SERIALSTATE	Contains information about the simulated serial connection.
USBH_CDC_RW_CONTEXT	Contains information which is passed to the USBH_CDC_ON_COMPLETE_FUNC user callback when using asynchronous write and read.

Table 9.26: emUSB-Host CDC data structure overview

9.3.1 USBH_CDC_DEVICE_INFO

Definition

```
typedef struct {  U16  VendorId;
                  U16  ProductId;
                  U8   acSerialNo[255];
} USBH_CDC_DEVICE_INFO;
```

Description

Contains information about a CDC compatible device.

Members

Member	Description
VendorId	Vendor identification number.
ProductId	Product identification number.
acSerialNo	Serial number string.

Table 9.27: USBH_CDC_DEVICE_INFO parameter list

9.3.2 USBH_CDC_SERIALSTATE

Definition

```
typedef struct {  U8 bRxCarrier;
                  U8 bTxCarrier;
                  U8 bBreak;
                  U8 bRingSignal;
                  U8 bFraming;
                  U8 bParity;
                  U8 bOverRun;
}  USBH_CDC_SERIALSTATE;
```

Description

Contains information about the simulated serial connection.

Members

Member	Description
bRxCarrier	State of receiver carrier detection mechanism of device. This signal corresponds to V.24 signal 109 and RS-232 signal DCD.
bTxCarrier	State of transmission carrier. This signal corresponds to V.24 signal 106 and RS-232 signal DSR.
bBreak	State of break detection mechanism of the device.
bRingSignal	State of ring signal detection of the device.
bFraming	A framing error has occurred.
bParity	A parity error has occurred.
bOverRun	Received data has been discarded due to overrun in the device.

Table 9.28: USBH_CDC_SERIALSTATE parameter list

9.3.3 USBH_CDC_RW_CONTEXT

Definition

```
typedef struct _USBH_CDC_RW_CONTEXT {
  void          * pUserContext;
  USBH_STATUS     Status;
  U32             NumBytesTransferred;
  void          * pUserBuffer;
  U32             UserBufferSize;
} USBH_CDC_RW_CONTEXT;
```

Description

Contains information which is passed to the USBH_CDC_ON_COMPLETE_FUNC user call-back when using asynchronous write and read.

When this structure is passed to USBH_CDC_ReadAsync() or USBH_CDC_WriteAsync() its member do not have to be initialized. The structure is used to pass information to the user callback.

The application should not modify the information in this structure.

Members

Member	Description
pUserContext	Pointer to a USBH_CDC_RW_CONTEXT structure.
Status	Status variable.
NumBytesTransferred	Number of bytes transfered.
pUserBuffer	Pointer to the user-provided buffer.
UserBufferSize	Size of the user buffer.

Table 9.29: USBH_CDC_RW_CONTEXT parameter list

9.4 Type definitions

This chapter describes the used structures defined in the header file `USBH_CDC.h`.

Type	Description
USBH_CDC_ON_COMPLETE_FUNC	Definition for the user callback used with the asynchronous read and write operations.

Table 9.30: emUSB-Host CDC type definition overview

9.4.1 USBH_CDC_ON_COMPLETE_FUNC

Definition

```
typedef void USBH_CDC_ON_COMPLETE_FUNC(USBH_CDC_RW_CONTEXT * pRWContext);
```

Description

Definition of the callback which has to be specified when using `USBH_CDC_ReadAsync()` or `USBH_CDC_WriteAsync()`.

Parameters

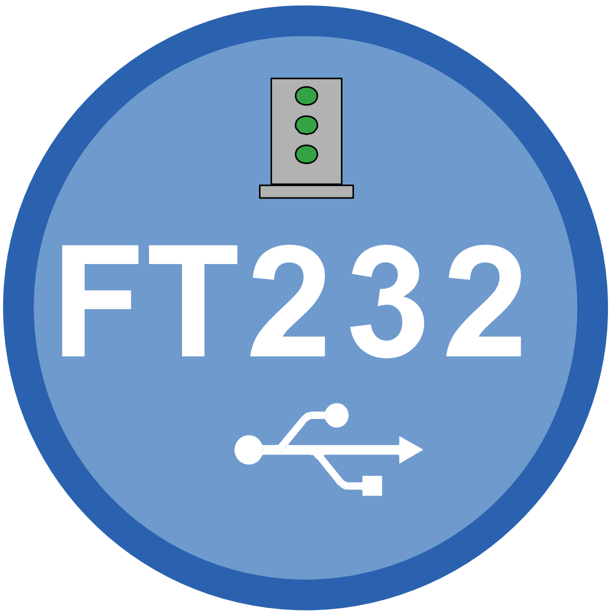
Paramter	Description
`pRWContext`	Pointer to a `USBH_CDC_RW_CONTEXT` structure.

Table 9.31: USBH_CDC_ON_COMPLETE_FUNC parameter list

Chapter 10

FT232 Device Driver (Add-On)

This chapter describes the optional emUSB-Host add-on "FT232 device driver". It allows communication with an FTDI FT232 USB device, typically serving as USB to RS232 converter.



10.1 Introduction

The FT232 driver software component of emUSB-Host allows the communication with FTDI FT232 devices. It implements the FT232 protocol specified by FTDI which is a vendor specific protocol. The protocol allows emulation of serial communication via USB.

This chapter provides an explanation of the functions available to application developers via the FT232 driver software. All the functions and data types of this add-on are prefixed with the "USBH_FT232_" text.

10.1.1 Overview

A FT232 device connected to the emUSB-Host is automatically configured and added to an internal list. If the FT232 driver has been registered, it is notfied via callback when a FT232 device has been added or removed. The driver then can notify the application program, when a callback function has been registered via `USBH_FT232_RegisterNotification()`. In order to communicate to a such a device, the application has to call the `USBH_FT232_Open()`, passing the device index. The FT232 devices are identified by an index. The first connected device gets assigned the index 0, the second index 1, and so on.

10.1.2 Features

The following features are provided:

- Compatibility with different FT232 devices.
- Ability to send and receive data.
- Ability to set various parameters, such as baudrate, number of stop bits, parity.
- Handling of multiple FT232 devices at the same time.
- Notifications about FT232 connection status.
- Ability to query the FT232 line and modem status.

10.1.3 Example code

An example application which uses the API is provided in the OS_USBH_FT232.c file. This example displays information about the FT232 device in the I/O terminal of the debugger. In addition the application then starts a simple echo server, sending back the received data.

10.1.4 Compatibility

The following devices work with the current FT232 driver:

- FT8U232AM
- FT232B
- FT232R
- FT2232D

10.1.5 Further reading

For more information about the FTDI FT232 devices, please take a look at the hardware manual and D2XX Programmer's Guide manual (Document Reference No.: FT_000071) available from www.ftdichip.com.

10.2 API Functions

This chapter describes the emUSB-Host FT232 driver API functions. These functions are defined in the header file USBH_FT232.h.

Function	Description
USBH_FT232_Init()	Initializes and registers the FT232 driver with the USB Host stack.
USBH_FT232_Exit()	Deinitializes the FT232 device driver.
USBH_FT232_RegisterNotification()	Sets a callback in order to be notified when a device is added or removed.
USBH_FT232_ConfigureDefaultTimeout()	Sets the default read and write timeout that shall be used when a new device is connected.
USBH_FT232_Open()	Opens a device given by an index.
USBH_FT232_Close()	Closes a handle to an opened device.
USBH_FT232_GetDeviceInfo()	Retrieves the information about the FT232 device.
USBH_FT232_ResetDevice()	Resets the FT232 device.
USBH_FT232_SetTimeouts()	Sets up the default timeouts the host waits until the data transfer is aborted.
USBH_FT232_Read()	Reads data from the FT232 device.
USBH_FT232_Write()	Writes data to the FT232 device.
USBH_FT232_AllowShortRead()	The configuration function allows to let the read function to return as soon as data are available.
USBH_FT232_SetBaudRate()	Sets the baud rate for the opened device.
USBH_FT232_SetDataCharacteristics()	Setups the serial communication with the given characteristics.
USBH_FT232_SetFlowControl()	Sets the flow control for the device.
USBH_FT232_SetDtr()	Sets the Data Terminal Ready (DTR) control signal.
USBH_FT232_ClrDtr()	Clears the Data Terminal Ready (DTR) control signal.
USBH_FT232_SetRts()	Sets the Request To Send (RTS) control signal.
USBH_FT232_ClrRts()	Clears the Request To Send (RTS) control signal.
USBH_FT232_GetModemStatus()	Gets the modem status and line status from the device.
USBH_FT232_SetChars()	Sets the special characters for the device.
USBH_FT232_Purge()	Purges receive and transmit buffers in the device.
USBH_FT232_GetQueueStatus()	Gets the number of bytes in the receive queue.
USBH_FT232_SetBreakOn()	Sets the BREAK condition for the device.
USBH_FT232_SetBreakOff()	Resets the BREAK condition for the device.
USBH_FT232_SetLatencyTimer()	Sets the latency timer value.

Table 10.1: emUSB-Host FT232 device driver API function overview

Function	Description
USBH_FT232_GetLatencyTimer()	Gets the current value of the latency timer.
USBH_FT232_SetBitMode()	Enables different chip modes.
USBH_FT232_GetBitMode()	Gets the instantaneous value of the data bus.

Table 10.1: emUSB-Host FT232 device driver API function overview (Continued)

10.2.1 USBH_FT232_Init()

Description

Initializes and registers the FT232 device driver to emUSB-Host.

Prototype

```
USBH_BOOL USBH_FT232_Init(void);
```

Return Value

`== 1`: Success
`== 0`: Could not register FT232 device driver

10.2.2 USBH_FT232_Exit()

Description

Unregisters and deinitializes the FT232 device driver from emUSB-Host.

Prototype

```
void USBH_FT232_Exit(void);
```

Additional information

This function will release resources that were used by this device driver. It has to be called if the application is closed. This has to be called before `USBH_Exit()` is called. No more functions of this module may be called after calling `USBH_FT232_Exit()`. The only exception is `USBH_FT232_Init()`, which would in turn reinitialize the module and allows further calls.

10.2.3 USBH_FT232_RegisterNotification()

Description

Sets a callback in order to be notified when a device is added or removed.

Prototype

```
void USBH_FT232_RegisterNotification(
      USBH_NOTIFICATION_FUNC * pfNotification,
      void                   * pContext);
```

Parameter

Parameter	Description
pfNotification	[IN] - Pointer to a function the stack should call when a device is connected or disconnected.
pContext	[IN] - Pointer to a user context that should be passed to the call-back function.

Table 10.2: USBH_FT232_RegisterNotification() parameter list

Additional Information

Only one notification function can be set for all devices. To unregister, call this function with the pfNotification parameter set to NULL.

Example

```
/*********************************************************************
*
*       _cbOnAddRemoveDevice
*
*  Function description
*    Callback, called when a device is added or removed.
*    Call in the context of the USBH_Task.
*    The functionality in this routine should not block
*/
static void _cbOnAddRemoveDevice(void * pContext, U8 DevIndex, USBH_DEVICE_EVENT
Event) {
  pContext = pContext;
  switch (Event) {
  case USBH_DEVICE_EVENT_ADD:
    printf("\n**** Device added\n");
    _DevIndex = DevIndex;
    _DevIsReady = 1;
    break;
  case USBH_DEVICE_EVENT_REMOVE:
    printf("\n**** Device removed\n");
    _DevIsReady = 0;
    _DevIndex   = -1;
    _Removed    = 1;
    break;
  default:;    // Should never happen
  }
}
```



```
/*********************************************************************
*
*       FT232_Task
*
*  Function description
*    This task shall handle FT232 devices. It initialize the FT232 driver
*    and sets a  notification callback in order to be informed about adding
*    removing of FT232 devices.
*/
void FT232_Task(void) {
  USBH_FT232_Init();
  USBH_FT232_RegisterNotification(_cbOnAddRemoveDevice, NULL);
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay(100);
    if (_DevIsReady) {
      _OnDevReady();
    }
  }
}
```


10.2.4 USBH_FT232_ConfigureDefaultTimeout()

Description

Sets the default read and write timeout that shall be used when a new device is connected.

Prototype

```
void USBH_FT232_ConfigureDefaultTimeout(U32 ReadTimeout,
                                        U32 WriteTimeout);
```

Parameter

Parameter	Description
ReadTimeout	[IN] - Default read timeout given in ms.
WriteTimeout	[IN] - Default write timeout given in ms.

Table 10.3: USBH_FT232_ConfigureDefaultTimeout() parameter list

Additional information

The function shall be called after USBH_FT232_Init() has been called, otherwise the behavior is undefined.

Example

```
void FT232_Task(void) {
  USBH_FT232_Init();
  USBH_FT232_ConfigureDefaultTimeout(50, 50); // Configure default timeout for read
                                              //and write to 50ms.
[...]
```


10.2.5 USBH_FT232_Open()

Description

Opens a device given by an index.

Prototype

```
USBH_FT232_HANDLE USBH_FT232_Open(unsigned Index);
```

Parameter

Parameter	Description
Index	[IN] - Index of the device that shall be opened. In general this means: the first connected device is 0, second device is 1 etc.

Table 10.4: USBH_FT232_Open() parameter list

Return Value

`== 0`: Device could not be opened (Removed or not available)
`!= 0`: Handle to the device

Example

```
USBH_FT232_HANDLE hDevice;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.6 USBH_FT232_Close()

Description

Closes a handle to an opened device.

Prototype

```
USBH_STATUS USBH_FT232_Close(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the device which shall be closed.

Table 10.5: USBH_FT232_Close() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE hDevice;
U32               NumBytesWritten;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  USBH_FT232_Write(hDevice, "Hello\n", 6, &NumBytesWritten);
  USBH_FT232_Close(hDevice);
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.7 USBH_FT232_GetDeviceInfo()

Description

Retrieves the information about the FT232 device.

Prototype

```
USBH_STATUS USBH_FT232_GetDeviceInfo(USBH_FT232_HANDLE        hDevice,
                                     USBH_FT232_DEVICE_INFO * pDevInfo);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pDevInfo	[OUT] - Pointer to a USBH_SFT232_DEVICE_INFO structure to store information related to the device.

Table 10.6: USBH_FT232_GetDeviceInfo() parameter list

Return Value

== USBH_STATUS_SUCCESS: Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  USBH_FT232_GetDeviceInfo(hDevice, &DeviceInfo);
  printf("Vendor  ID = 0x%4.4x\n"
         "Product ID = 0x%4.4x\n"
         "bcdDevice  = 0x%4.4x\n", DeviceInfo.VendorId,
                                   DeviceInfo.ProductId,
                                   DeviceInfo.bcdDevice);
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.8 USBH_FT232_ResetDevice()

Description

Resets the FT232 device.

Prototype

```
USBH_STATUS USBH_FT232_ResetDevice(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.7: USBH_FT232_ResetDevice() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_ResetDevice(hDevice); // Do the reset
  if (Status == USBH_STATUS_SUCCESS) {
    printf("Resetting the device was successful!\n");
  } else {
    printf("Failed to reset the device, Error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```

10.2.9 USBH_FT232_SetTimeouts()

Description

Sets up the timeouts the host waits until the data transfer is aborted.

Prototype

```
USBH_STATUS USBH_FT232_SetTimeouts(USBH_FT232_HANDLE hDevice,
                                   U32               ReadTimeout,
                                   U32               WriteTimeout);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
ReadTimeout	[IN] - Read timeout given in ms.
WriteTimeout	[IN] - Write timeout given in ms.

Table 10.8: USBH_FT232_SetTimeouts() parameter list

Return Value

== USBH_STATUS_SUCCESS Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_SetTimeouts(hDevice, 30, 30); // Set timeout for both to 30ms.
  if (Status == USBH_STATUS_SUCCESS) {
    printf("Setting the timeout was successful!\n");
  } else {
    printf("Failed to set timeout, Error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.10 USBH_FT232_Read()

Description

Reads data from the FT232 device.

Prototype

```
USBH_STATUS USBH_FT232_Read(USBH_FT232_HANDLE hDevice,
                            U8   *            pData,
                            U32               NumBytes,
                            U32 *             pNumBytesRead);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pData	[OUT] - Pointer to the buffer that receives the data from the device.
NumBytes	[IN] - Number of bytes to be read from the device.
pNumBytesRead	[OUT] - Pointer to a variable of type U32 which receives the number of bytes read from the device.

Table 10.9: USBH_FT232_Read() parameter list

Return Value

`== USBH_STATUS_SUCCESS` Success
Any other value means error.

Additional information

USBH_FT232_Read() always returns the number of bytes read in pNumBytesRead.
This function does not return until NumBytes bytes have been read into the buffer unless short read mode is enabled. This allows USBH_FT232_Read() to return when either data have been read from the queue or as soon as some data have been read from the device.
The number of bytes in the receive queue can be determined by calling USBH_FT232_GetQueueStatus(), and passed to USBH_FT232_Read() as NumBytes so that the function reads the device and returns immediately.
When a read timeout value has been specified in a previous call to USBH_FT232_SetTimeouts(), USBH_FT232_Read() returns when the timer expires or NumBytes have been read, whichever occurs first. If the timeout occurs, USBH_FT232_Read() reads available data into the buffer and returns USBH_STATUS_TIMEOUT.
An application should use the function return value and pNumBytesRead when processing the buffer. If the return value is USBH_STATUS_SUCCESS, and pNumBytesRead is equal to NumBytes then USBH_FT232_Read has completed normally.
If the return value is USBH_STATUS_TIMEOUT, pNumBytesRead may be less or even 0, in any case, pData will be filled with pNumBytesRead.
Any other return value suggests an error in the parameters of the function, or a fatal error like a USB disconnect.

10.2.11 USBH_FT232_Write()

Description

Writes data to the FT232 device.

Prototype

```
USBH_STATUS USBH_FT232_Write(USBH_FT232_HANDLE hDevice,
                             const U8 *        pData,
                             U32               NumBytes,
                             U32 *             pNumBytesWritten);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pData	[IN] - Pointer to the buffer that contains the data to be written to the device.
NumBytes	[IN] - Number of bytes to write to the device.
pNumBytesWritten	[OUT] - Pointer to a variable of type U32 which receives the number of bytes written to the device.

Table 10.10: USBH_FT232_Write() parameter list

Return Value

`== USBH_STATUS_SUCCESS` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS           Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_Write(hDevice, "SEGGER", 7); // Write SEGGER\0 over FT232
  if (Status == USBH_STATUS_SUCCESS) {
    printf("Write was successful!\n");
  } else {
    printf("Failed to write, Error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.12 USBH_FT232_AllowShortRead()

Description

The configuration function allows the read functions to return as soon as data is available.

Prototype

```
USBH_STATUS USBH_FT232_AllowShortRead(USBH_FT232_HANDLE hDevice,
                                      U8                AllowShortRead);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
AllowShortRead	[IN] - Define whether short read mode shall be used or not. 1 - Enable short read mode 0 - Disable short read mode.

Table 10.11: USBH_FT232_AllowShortRead() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

USBH_FT232_AllowShortRead() sets the USBH_FT232_Read() into a special mode - short read mode. When this mode is enabled, the function returns as soon as data has been read from the device. This allows the application to read data where the number of bytes to read is undefined.
To disable this mode, AllowShortRead shall be 0.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;
U32                    NumBytesWritten;
U32                    NumBytes2Write = 6;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_Write(hDevice, "Hello\n", NumBytes2Write, &NumBytesWritten);
  if (Status == USBH_STATUS_SUCCESS) {
    printf("All bytes have been written!\n");
  } else {
      printf("Not all bytes (%d of %d) have been written, error code = 0x%x",
NumBytesWritten, NumBytes2Write, Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.13 USBH_FT232_SetBaudRate()

Description

Sets the baud rate for the opened device.

Prototype

```
USBH_STATUS USBH_FT232_SetBaudRate(USBH_FT232_HANDLE hDevice,
                                   U32               Baudrate);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Baudrate	[IN] - Baud rate to set.

Table 10.12: USBH_FT232_SetBaudRate() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_SetbaudRate(hDevice, 115200);
  if (Status == USBH_STATUS_SUCCESS) {
    printf("SetBaudrate was successful!\n");
  } else {
    printf("Could not set baudrate, error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.14 USBH_FT232_SetDataCharacteristics()

Description

Sets up the serial communication with the given characteristics.

Prototype

```
USBH_STATUS USBH_FT232_SetDataCharacteristics(USBH_FT232_HANDLE hDevice,
                                              U8                Length,
                                              U8                StopBits,
                                              U8                Parity);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Length	[IN] - Number of bits per word: - must be USBH_FT232_BITS_8 or USBH_FT232_BITS_7
StopBits	[IN] - Number of stop bits - must be USBH_FT232_STOP_BITS_1 or USBH_FT232_STOP_BITS_2.
Parity	[IN] - Parity - must be one of the following values: USBH_FT232_PARITY_NONE USBH_FT232_PARITY_ODD USBH_FT232_PARITY_EVEN USBH_FT232_PARITY_MARK USBH_FT232_PARITY SPACE.

Table 10.13: USBH_FT232_SetDataCharacteristics() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_SetDataCharacteristics(hDevice,
                                             USBH_FT232_BITS_8,
                                             USBH_FT232_STOP_BITS_1,
                                             USBH_FT232_PARITY_NONE);
  if (Status == USBH_STATUS_SUCCESS) {,
    printf("Commnication options have been set successfully!\n");
  } else {
    printf("Could not set communication options, error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.15 USBH_FT232_SetFlowControl()

Description

Sets the flow control for the device.

Prototype

```
USBH_STATUS USBH_FT232_SetFlowControl(USBH_FT232_HANDLE hDevice,
                                      U16               FlowControl,
                                      U8                XonChar,
                                      U8                XoffChar);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
FlowControl	[IN] - Must be one of the following values: USBH_FT232_FLOW_NONE USBH_FT232_FLOW_RTS_CTS USBH_FT232_FLOW_DTR_DSR USBH_FT232_FLOW_XON_XOFF
XonChar	[IN] - Character that shall be used to signal Xon. Only used if flow control is USBH_FT232_FLOW_XON_XOFF.
XoffChar	[IN] - Character that shall be used to signal Xoff. Only used if flow control is USBH_FT232_FLOW_XON_XOFF.

Table 10.14: USBH_FT232_SetFlowControl() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Example

```
USBH_FT232_HANDLE      hDevice;
USBH_FT232_DEVICE_INFO DeviceInfo;
USBH_STATUS            Status;

hDevice = USBH_FT232_Open(0);  // Open device with index 0.
if (hDevice) {
  // Got a valid device handle
  Status = USBH_FT232_SetFlowControl(hDevice,
                                     USBH_FT232_FLOW_NONE,
                                                        0,
                                                        0);
  if (Status == USBH_STATUS_SUCCESS) {,
    printf("FlowControl have been set successfully!\n");
  } else {
    printf("Could not set flow control, error code = 0x%x", Status);
  }
} else {
  // Failed to open device, the device may be unavailable or was previously removed.
  printf("Failed to open device\n");
}
```


10.2.16 USBH_FT232_SetDtr()

Description

Sets the Data Terminal Ready (DTR) control signal.

Prototype

```
USBH_STATUS USBH_FT232_SetDtr(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.15: USBH_FT232_SetDtr() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.17 USBH_FT232_ClrDtr()

Description

Clears the Data Terminal Ready (DTR) control signal.

Prototype

```
USBH_STATUS USBH_FT232_ClrDtr(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.16: USBH_FT232_ClrDtr() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.18 USBH_FT232_SetRts()

Description

Sets the Request To Send (RTS) control signal.

Prototype

```
USBH_STATUS USBH_FT232_SetRts(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.17: USBH_FT232_SetRts() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.19 USBH_FT232_ClrRts()

Description

Clears the Request To Send (RTS) control signal.

Prototype

```
USBH_STATUS USBH_FT232_ClrRts(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.18: USBH_FT232_ClrRts() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.20 USBH_FT232_GetModemStatus()

Description

Gets the modem status and line status from the device.

Prototype

```
USBH_STATUS USBH_FT232_GetModemStatus(USBH_FT232_HANDLE hDevice,
                                      U32 *             pModemStatus);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pModemStatus	[IN] - Pointer to a variable of type U32 which receives the modem status and line status from the device.

Table 10.19: USBH_FT232_GetModemStatus() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

The least significant byte of the `pModemStatus` value holds the modem status.
The line status is held in the second least significant byte of the `pModemStatus` value.

The modem status is bit-mapped as follows:

```
Clear To Send        (CTS) = 0x10,
Data Set Ready       (DSR) = 0x20,
Ring Indicator       (RI)  = 0x40,
Data Carrier Detect  (DCD) = 0x80.
```

The line status is bit-mapped as follows:

```
Overrun Error        (OE)  = 0x02,
Parity Error         (PE)  = 0x04,
Framing Error        (FE)  = 0x08,
Break Interrupt      (BI)  = 0x10.
TxHolding register empty   = 0x20
TxEmpty                    = 0x40
```

10.2.21 USBH_FT232_SetChars()

Description

Sets the special characters for the device.

Prototype

```
USBH_STATUS USBH_FT232_SetChars(USBH_FT232_HANDLE hDevice,
                                U8                EventChar,
                                U8                EventCharEnabled,
                                U8                ErrorChar,
                                U8                ErrorCharEnabled);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
EvenChar	[IN] - Event character
EventCharEnable	[IN] - 0 if event character disabled, non-zero otherwise.
ErrorChar	[IN] - Error character
ErrorCharEnabled	[IN] - 0 if error character disabled, non-zero otherwise.

Table 10.20: USBH_FT232_SetChars() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

This function allows to insert special characters in the data stream to represent events triggering or errors occurring.

10.2.22 USBH_FT232_Purge()

Description

Purges receive and transmit buffers in the device.

Prototype

```
USBH_STATUS USBH_FT232_Purge(USBH_FT232_HANDLE hDevice,
                             U32               Mask);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Mask	[IN] - Combination of USBH_FT232_PURGE_RX and USBH_FT232_FT_PURGE_TX.

Table 10.21: USBH_FT232_Purge() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.23 USBH_FT232_GetQueueStatus()

Description

Gets the number of bytes in the receive queue.

Prototype

```
USBH_STATUS USBH_FT232_GetQueueStatus(USBH_FT232_HANDLE hDevice,
                                      U32 *             pRxBytes);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pRxBytes	[OUT] - Pointer to a variable of type U32 which receives the number of bytes in the receive queue.

Table 10.22: USBH_FT232_GetQueueStatus() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.24 USBH_FT232_SetBreakOn()

Description

Sets the BREAK condition for the device.

Prototype

```
USBH_STATUS USBH_FT232_SetBreakOn(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.23: USBH_FT232_SetBreakOn() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.25 USBH_FT232_SetBreakOff()

Description

Resets the BREAK condition for the device.

Prototype

```
USBH_STATUS USBH_FT232_SetBreakOff(USBH_FT232_HANDLE hDevice);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.24: USBH_FT232_SetBreakOff() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

10.2.26 USBH_FT232_SetLatencyTimer()

Description

Sets the latency timer value.

Prototype

```
USBH_STATUS USBH_FT232_SetLatencyTimer(USBH_FT232_HANDLE hDevice,
                                       U8                Latency);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Latency	[IN] - Required value, in milliseconds, of latency timer. Valid range is 2 – 255.

Table 10.25: USBH_FT232_SetLatencyTimer() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

In the FT8U232AM and FT8U245AM devices, the receive buffer timeout that is used to flush remaining data from the receive buffer was fixed at 16 ms. Therefore this function cannot be used with these devices.
In all other FTDI devices, this timeout is programmable and can be set at 1 ms intervals between 2ms and 255 ms.
This allows the device to be better optimized for protocols requiring faster response times from short data packets.

10.2.27 USBH_FT232_GetLatencyTimer()

Description

Gets the current value of the latency timer.

Prototype

```
USBH_STATUS USBH_FT232_GetLatencyTimer(USBH_FT232_HANDLE hDevice,
                                       U8 *              pLatency);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.

Table 10.26: USBH_FT232_GetLatencyTimer() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

Please refer to `USBH_FT232_SetLatencyTimer()` for more information about latency timer.

10.2.28 USBH_FT232_SetBitMode()

Description

Enables different chip modes.

Prototype

```
USBH_STATUS USBH_FT232_SetBitMode(USBH_FT232_HANDLE hDevice,
                                  U8                Mask,
                                  U8                Enable);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
Mask	[IN] - Required value for bit mode mask. This sets up which bits are inputs and outputs. A bit value of 0 sets the corresponding pin to an input. A bit value of 1 sets the corresponding pin to an output. In the case of CBUS Bit Bang, the upper nibble of this value controls which pins are inputs and outputs, while the lower nibble controls which of the outputs are high and low.
ucMode	[IN] - Mode value. Can be one of the following value: 0x00 = Reset 0x01 = Asynchronous Bit Bang 0x02 = MPSSE (FT2232, FT2232H, FT4232H and FT232H devices only) 0x04 = Synchronous Bit Bang (FT232R, FT245R, FT2232, FT2232H, FT4232H and FT232H devices only) 0x08 = MCU Host Bus Emulation Mode (FT2232, FT2232H, FT4232H and FT232H devices only) 0x10 = Fast Opto-Isolated Serial Mode (FT2232, FT2232H, FT4232H and FT232H devices only) 0x20 = CBUS Bit Bang Mode (FT232R and FT232H devices only) 0x40 = Single Channel Synchronous 245 FIFO Mode (FT2232H and FT232H devices only)

Table 10.27: USBH_FT232_SetBitMode() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

For further information please refer to the HW-reference manuals and application note on the FTDI website.

10.2.29 USBH_FT232_GetBitMode()

Description

Gets the instantaneous value of the data bus.

Prototype

```
USBH_STATUS USBH_FT232_GetBitMode(USBH_FT232_HANDLE hDevice,
                                  U8 *              pMode);
```

Parameter

Parameter	Description
hDevice	[IN] - Handle to the opened device.
pMode	[OUT] - Pointer to U8 to store the instantaneous data bus value.

Table 10.28: USBH_FT232_GetBitMode() parameter list

Return Value

`== USBH_STATUS_SUCCESS:` Success
Any other value means error.

Additional information

For further information please refer to the HW-reference manuals and application note on the FTDI website.

Chapter 11

Configuring emUSB-Host

emUSB-Host can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values which matching the requirements of most applications. Network interface drivers can be added at runtime.

The default configuration of emUSB-Host can be changed via compile-time flags which can be added to `USBH_Conf.h`. This is the main configuration file for the emUSB-Host stack.

11.1 Runtime configuration

Every driver folder includes a configuration file with implementations of runtime configuration function explained in this chapter. This function can be customized. Functions that can be called within this fucntion are described later in this chapter.

11.1.1 USBH_X_Config()

Description

Helper function to prepare and configure the emUSB-Host stack.

Prototype

```
void USBH_X_Config(void);
```

Additional information

This function is called by the startup code of the emUSB-Host stack from USBH_Init(). This is the place where you can add and configure the hardware driver.

Example

```
/*********************************************************************
*
*       USBH_X_Config
*
*  Function description
*    Configures the emUSB-Host for an AT91SAM9G45 target.
*/

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((void *)((ALLOC_BASE + 0xff) & ~0xffUL), ALLOC_SIZE);

  // Default values: 2 Devices are supported.
  // USBH_ConfigMaxUSBDevices       (2);
  // Default values: 2 Bulk, 2 Interrupt, 0 Isochronous endpoints are supported.
  // USBH_ConfigMaxNumEndpoints     (2, 2, 0);
  // Default values: The hub module is disabled, this is done to save memory.
  // USBH_ConfigSupportExternalHubs (0);
  // Default values: 300 ms before the host starts communicating with the device.
  // USBH_ConfigPowerOnGoodTime     (300);

  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  //       of your communication, since the debugger stops the target
  //       for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0xFFFFFFFF); // 0xFFFFFFFF: Do not filter: Output all warnings.
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    | USBH_MTYPE_HID
                   );
  BSP_USBH_Init();
  USBH_OHC_Add((void*)OHCI_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```

11.2 Configuration functions

These functions must only be called within the USBH_X_Config() function.
These functions configure and add a specific USB host driver to emUSB Host stack.
.

Function	Description
USBH_AssignMemory()	Configures a memory pool for emUSB-Host internal handling.
USBH_AssignTransferMemory()	Configures a memory pool for the data exchange with the host controller.
USBH_ConfigTransferBufferSize()	Configures the size of copy buffer that are used if the USB controller has limited access to the system memory.
USBH_ConfigRootHub()	Configures some features of the controller's root hub.
USBH_ConfigMaxUSBDevices()	Configures the number of devices that shall be used.
USBH_ConfigMaxNumEndpoints()	Configures the number of endpoints that shall be used.
USBH_ConfigSupportExternalHubs()	Specifies whether USB hub shall be supported or not.
USBH_ConfigPowerOnGoodTime()	Specifies the timeout value before the start of communication.
USBH_OHCI_Add()	Adds an OHCI compliant host controller to emUSB Host.
USBH_STM32_Add()	Adds an ST STM32 10x,20x,21x compliant full-speed host controller to emUSB Host.
USBH_RX62_Add()	Adds an Renesas RX62x compliant full-speed host controller to emUSB Host.
USBH_AVR32_Add()	Adds an Atmel AVR32 full-speed compliant host controller to emUSB Host.
USBH_AVR32_ConfigureEPRAM()	Tells the Atmel AVR32 emUSB-Host driver the location of the EP-RAM.
USBH_EHCI_Add()	Adds an EHCI compliant host controller to emUSB Host.
USBH_EHCI_Config_SetM2MEndianMode()	Sets up the internal EHCI memory-2-memory transfer endian mode.

Table 11.1: emUSB-Host Configuration API function overview

11.2.1 USBH_AssignMemory()

Description

Sets up storage for the memory allocator.

Prototype

```
void USBH_AssignMemory(U32 * pMem, U32 NumBytes);
```

Parameter

Parameter	Description
pMem	Pointer to a caller allocated memory area.
NumBytes	Size of memory area in bytes.

Table 11.2: USBH_AssignMemory() parameter list

Additional information

emUSB-Host comes with its own dynamic memory allocator optimized for its needs. You can use this function to set up up a memory area for the heap. The best place to call it is in the USBH_X_Config() function.

In cases where the USB host controller has limited access to system memory, the USBH_AssignTransferMemory() must be additionally called.

11.2.2 USBH_AssignTransferMemory()

Description

Assign memory used for DMA transfers of the USB-Host controller.
In order to use this memory, the following requirements must be fulfilled:

- Physical address == virtual address (no address translation by an MMU).
- Not cachable/bufferable.
- “fast” access to avoid timeouts.
- USB-Host controller muss have full read/write access.

Prototype

```
void USBH_AssignTransferMemory(U32 * pMem, U32 NumBytes);
```

Parameter

Parameter	Description
pMem	Pointer to a memory area.
NumBytes	Size of memory area in bytes.

Table 11.3: USBH_AssignTransferMemory() parameter list

Additional information

Use of this function is required only in systems in which "normal" default memory does not fulfill all of these criteria.
In simple microcontroller systems without cache, MMU and external RAM, use of this function is not required. If no transfer memory is assigned, memory assigned with USBH_AssignMemory() is used instead.
This function is normally used with an OHCI controller.

11.2.3 USBH_ConfigTransferBufferSize()

Description

Configures the size of copy buffer that is used if the USB controller has limited access to the system memory.

Prototype

```
void USBH_ConfigTransferBufferSize(U32 Size);
```

Parameter

Parameter	Description
Size	Size of copy transfer buffer given in bytes.

Table 11.4: USBH_ConfigTransferBufferSize() parameter list

Additional information

This function is normally used with an OHCI controller.

11.2.4 USBH_ConfigRootHub()

Description

Sets up additional storage for the memory allocator. The USB host controller must have read/write access to the configured memory area.

Prototype

```
void USBH_ConfigRootHub(U8 SupportOvercurrent,
                        U8 PortsAlwaysPowered,
                        U8 PerPortPowered);
```

Parameter

Parameter	Description
SupportOvercurrent	If overcurrent is supported by the host controller, SupportOvercurrent needs to be set to 1.
PortsAlwaysPowered	Specifies whether the USB port is always powered.
PerPortPowered	Specifies that the power of each port can be set individually.

Table 11.5: USBH_ConfigRootHub() parameter list

Additional information

Currently this function can only be used with the OHCI USB host controllers.

11.2.5 USBH_ConfigMaxUSBDevices()

Description

Configures the number of devices that shall be used.

Prototype

```
void USBH_ConfigMaxUSBDevices(U8 NumDevices);
```

Parameter

Parameter	Description
NumDevices	Sets the number of devices that shall be handled.

Table 11.6: USBH_AssignTransferMemory() parameter list

Additional information

Depending on how many devices shall be connected to the USB device, this configuration function may help to reduce the RAM consumption.
In order to know how many devices shall be connected to the host, please note that a USB hub is also a USB device which needs to be managed.

The default value is set to 2 devices.

11.2.6 USBH_ConfigMaxNumEndpoints()

Description

Configures the number of devices that shall be used.

Prototype

```
void USBH_ConfigMaxUSBDevices(U8 MaxNumBulkEndpoints,
                              U8 MaxNumIntEndpoints,
                              U8 MaxNumIsoEndpoints);
```

Parameter

Parameter	Description
MaxNumBulkEndpoints	Sets the number of bulk endpoints that shall be handled.
MaxNumIntEndpoints	Sets the number of interrupt endpoints that shall be handled.
MaxNumIsoEndpoints	Sets the number of isochrounous endpoints that shall be handled.

Table 11.7: USBH_AssignTransferMemory() parameter list

Additional information

The value depends on the device and classes that shall be used with emUSB Host. The following table shows some classes where the endpoint information is fixed or recommended:

Device class	Number of endpoints
MSD	2 Bulk endpoints.
HID	1 or 2 interrupt endpoints.
Printer	2 bulk endpoints.
CDC	2 bulk endpoints and 1 interrupt endpoint.
Hub	1 interrupt endpoint.

The default values are - 2 Bulk endpoints, 2 Interrupt endpoints, 0 Isochronous endpoints.

11.2.7 USBH_ConfigSupportExternalHubs()

Description

Specifies whether USB hub shall be supported or not.

Prototype

```
void USBH_ConfigSupportExternalHubs (U8 OnOff);
```

Parameter

Parameter	Description
OnOff	1 - Enable support for USB hubs. 0 - Disable support for USB hubs.

Table 11.8: USBH_AssignMemory() parameter list

Additional information

The default value is 0 in order to save memory by not linking the hub module.

11.2.8 USBH_ConfigPowerOnGoodTime()

Description

Configures the default power on time that the host waits for before starting to communicate with the device.

Prototype

```
void USBH_ConfigPowerOnGoodTime(unsigned PowerGoodTime);
```

Parameter

Parameter	Description
PowerGoodTime	The time to wait in ms.

Table 11.9: USBH_AssignMemory() parameter list

Additional information

If you are dealing with problematic devices which have long initialization sequences it is advisable to increase this timeout.

The default value is 300 ms.

11.2.9 USBH_OHCI_Add()

Description

Adds an OHCI compliant host controller to emUSB Host.

Prototype

```
U32 USBH_OHCI_Add(void * pBase);
```

Parameter

Parameter	Description
pBase	Base address of the OHCI controller.

Table 11.10: USBH_AssignTransferMemory() parameter list

Additional information

This functions checks and adds the USB host controller to the stack. Please make sure that an add-function is only called once.

11.2.10 USBH_STM32_Add()

Description

Adds a STM32 full-speed compliant host controller to emUSB Host.

Prototype

```
U32 USBH_STM32_Add(void * pBase);
```

Parameter

Parameter	Description
pBase	Base address of the USB controller.

Table 11.11: USBH_AssignTransferMemory() parameter list

Additional information

This functions checks and adds the USB host controller to the stack. Please make sure that an add-function is only called once.

11.2.11 USBH_RX62_Add()

Description

Adds a Renesas RX62 full-speed compliant host controller to emUSB Host.

Prototype

```
U32 USBH_RX62_Add(void * pBase);
```

Parameter

Parameter	Description
pBase	Base address of the USB controller.

Table 11.12: USBH_AssignTransferMemory() parameter list

Additional information

This functions checks and adds the USB host controller to the stack. Please make sure that an add-function is only called once.

11.2.12 USBH_AVR32_Add()

Description

Adds an Atmel AVR32 full-speed compliant host controller to emUSB Host.

Prototype

```
U32 USBH_AVR32_Add(void * pBase);
```

Parameter

Parameter	Description
pBase	Base address of the USB controller.

Table 11.13: USBH_AssignTransferMemory() parameter list

Additional information

This functions checks and adds the USB host controller to the stack. Please make sure that an add-function is only called once.

11.2.13 USBH_AVR32_ConfigureEPRAM()

Description

Tells the Atmel AVR32 emUSB-Host driver the location of the EP-RAM.

Prototype

```
U32 USBH_AVR32_ConfigureEPRAM(U32 RamBase);
```

Parameter

Parameter	Description
RamBase	Base address of the EP-RAM.

Table 11.14: USBH_AVR32_ConfigureEPRAM() parameter list

Additional information

After USBH_AVR32_Add has been called, this is the next function that shall be called. This function will then tell the AVR32 driver where the pipe (endpoint) RAM is located, otherwise there will be no communition between Host and Device.

11.2.14 USBH_EHCI_Add()

Description

Adds an EHCI compliant host controller to emUSB Host.

Prototype

```
U32 USBH_EHCI_Add(void * pBase);
```

Parameter

Parameter	Description
pBase	Base address of the EHCI controller.

Table 11.15: USBH_AssignTransferMemory() parameter list

Additional information

This function checks and adds the USB host controller to the stack. Please make sure that an add-function is only called once.

11.2.15 USBH_EHCI_Config_SetM2MEndianMode()

Description

Sets up the internal EHCI memory-2-memory transfer endian mode.

Prototype

```
void USBH_EHCI_Config_SetM2MEndianMode(U32 HCIndex, int Endian);
```

Parameter

Parameter	Description
HCIndex	HostController Index returned by the USBH_EHCI_Add() function.
Endian	Two values are allowed: USBH_EHCI_M2M_ENDIAN_MODE_LITTLE -> use little endian mode for memory-2-memory (DMA) transfers. USBH_EHCI_M2M_ENDIAN_MODE_BIG -> use big endian mode for memory-2-memory (DMA) transfers.

Table 11.16: USBH_EHCI_Config_SetM2MEndianMode() parameter list

Additional information

This only has an effect on the DMA transfers. Both SFRs and DMA descriptors are still in the endian defined by MCU/EHCI controller manufacturer.
In normal cases the SFRs and DMA descriptors are in CPU native endian mode.

11.3 Compile-time configuration

The following types of configuration macros exist:

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

Function replacements "F"

Macros can basically be treated like regular functions although certain limitations apply, as a macro is still put into the code as simple text replacement. Function replacements are mainly used to add specific functionality to a module which is highly hardware-dependent. This type of macro is always declared using brackets (and optional parameters).

11.3.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
		Debug macros	
N	USBH_DEBUG	0	emUSB-Host can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, emUSB-Host uses the logging routines. These routines can be blank, they are not required for the functionality of emUSB-Host. In a target system, they are typically not required in a release (production) build, a production build typically uses a lower debug level. The following table lists the values USBH_DEBUG define can take:
			0 - Used for release builds. Includes no debug options.
			1 - Used in debug builds to include support for "panic" checks.
			2 - Used in debug builds to include warning, log messages and "panic" checks.
		Optimization macros	
F	USBH_MEMCPY	memcpy (routine in standard C-library)	Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized version for the IAR compiler is supplied.
F	USBH_MEMSET	memset (routine in standard C-library)	Macro to define an optimized memset routine to speed up the stack. An optimized memset routine is typically implemented in assembly language.
F	USBH_MEMCMP	memcmp (routine in standard C-library)	Macro to define an optimized memcmp routine to speed up the stack. An optimized memcmp routine is typically implemented in assembly language.

Chapter 12

Host controller specifics

This chapter describes some specific information about the host controller and its configuration with emUSB-Host.

12.1 Introduction

A emUSB-Host driver is responsible for handling all communication between the generic upper layer of emUSB-Host and the USB controller hardware.
Specifically the following items are handled by the driver:

- Enabling/Disabling the USB host controller.
- Handling the host controller’s interrupts.
- Enabling/Disabling ports.
- Detecting USB devices.
- Resetting USB devices.
- Managing endpoints of all devices.
- Managing data transfers from the upper layer.

For emUSB-Host different USB host controller drivers can be delivered. Normally, the drivers are ready and do not need to be configured at all. Some controllers may need to be configured in a special manner, due to some limitation of the controller.

This chapters lists the controllers which require special configuration and describes how to configure those controllers.

12.2 Host Controller Drivers

The following drivers are available for emUSB-Host:

Driver	Description
OHCI Driver	This driver can be used for all OHCI compliant devices: Currently this driver has been tested with the following devices. Atmel AT91SAM9260 Atmel AT91SAM9261 Atmel AT91SAM9263 Atmel AT91SAM92G10 Atmel AT91SAM92G20 Atmel AT91SAM9XE Atmel AT91SAM92G45/M10 NXP LPC1754 NXP LPC1756 NXP LPC1758 NXP LPC1759 NXP LPC1765 NXP LPC1766 NXP LPC1768 NXP LPC1776 NXP LPC1777 NXP LPC1778 NXP LPC1785 NXP LPC1786 NXP LPC1787 NXP LPC1788 NXP LPC2387 NXP LPC2388 NXP LPC2420 NXP LPC2458 NXP LPC2460 NXP LPC2468 NXP LPC2470 NXP LPC2478 NXP LPC3180 NXP LPC3220 NXP LPC3230 NXP LPC3240 NXP LPC3250 NXP LH7A400 NXP LH7A404 Renesas V850ES Jx(G/H)3-U Toshiba TMPA900
ST STM32 Driver	Currently the following devices are supported by this driver. ST STM32F105 ST STM32F107
ST STM32F2_FS Driver	Currently the following devices are supported by this driver. ST STM32F207 ST STM32F217 ST STM32F407 ST STM32F417
ST STM32F2_HS Driver	Currently the following devices are supported by this driver. ST STM32F207 ST STM32F217 ST STM32F407 ST STM32F417

Driver	Description
Renesas RX Driver	This driver supports all Renesas RX62 MCU with integrated USB OTG controller. In some MCU two USB OTG controllers are available, both controllers can be used. It has been tested with the following MCUs. Renesas RX62N Renesas RX621 Renesas RX63N Renesas RX631.
Atmel AVR32 Driver	This driver supports all Atmel AVR32 UC3A/UC3B MCU with integrated USB OTG controller. It has been tested with the following MCUs. Atmel AT32UC3A0512-U.
Freescale Kinetis Full-Speed Driver	This driver supports all Freescale Kinetis Kxx/KLxx with integrated USB OTG controller. It has been tested with the following MCUs. Freescale K.
EHCI Driver	This driver can be used for all EHCI compliant devices. Currently this driver has been tested with the following devices. NXP LPC18xx NXP LPC43xx Freescale Kinetis K60 Freescale Kinetis K70 Atmel AT91SAM9G45/M10 Atmel AT91SAMA5D3x NXP LPC 3131 ColdFire MCF54415

12.3 General Information

In general many devices need to configure GPIO pins in order to use them with the USB host controller. In most cases the following pins are necessary:

- USB D+
- USB D-
- USB VBUS
- USB GND
- USB PowerOn
- USB OverCurrent

Please note that those pins need to be initialized within the `USBH_X_Config()` function. This has to be done before the USBH_Hostcontroller driver Add-function is called.
Another step that needs to be done before calling the USBH_Hostcontroller driver Add-function is initializing the clock for USB.

12.4 OHCI Driver

This driver can handle all USB transfers - this means it can handle Control, Interrupt, Bulk and Isochronous transfers. It handles up to 127 devices and has an automatic error detection.
Some OHCI implementations do not implement all features of OHCI. The individual limitations and configuration are described here for each tested implementation.

12.4.1 General information

For MCUs with an internal data cache controller such as ARM9/ARM11/Cortex A8 etc., please note the following:
In order to avoid cache inconsistency the allocation pool for emUSB-Host should be in a non-cached,non-buffered RAM region.

12.4.2 Atmel

Currently all tested Atmel OHCI implementations work flawlessly with the OHCI driver.

Configuration example:

```
#define ALLOC_SIZE         0x10000   // Size of memory dedicated to the stack in bytes
#define OHCI_BASE_ADDRESS  0x00500000

#define ALLOC_BASE    (((U32)&_aPool[0]) + 0x4000000)// Use the non cached SDRAM area
static U32 _aPool[((ALLOC_SIZE + 256) / 4)];       // Memory area used by the stack.


void USBH_X_Config(void) {
   USBH_AssignMemory((void *)((ALLOC_BASE + 0xff) & ~0xff), ALLOC_SIZE);      //
Assigning memory should be the first thing
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                      | USBH_MTYPE_INIT
                      | USBH_MTYPE_HID
                      | USBH_MTYPE_MSD
                      | USBH_MTYPE_APPLICATION
                     );
  USBH_SetLogFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_APPLICATION
                    );
  BSP_USBH_Init();
  USBH_OHC_Add((void*)OHCI_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```

12.4.3 NXP

All above listed NXP MCUs have a fully integrated OHCI-compliant host controller.
It supports all kind of USB transfer types, but has a limited access to memory. The OHCI controller has only access to the so called USB-RAM. This means that a transfer memory has to be specified. Within this transfer memory all relevant transfer descriptors and transfer memories are built. Typically 16kBytes are available for USB RAM, this limits the maximum connected devices to approx. 3-4.
How many root-hub ports are configured is automatically detected.

Configuration example:

```
#define ALLOC_SIZE           0x5800 // Size of memory dedicated to the stack in bytes
#define OHCI_BASE_ADDRESS    0x5000C000
#define TRANSFER_MEMORY_BASE 0x20080000 // Startaddress of the internal 16k USB SRAM
                                        // - AHB SRAM bank 1 is used
                                        // AHB SRAM bank 0 is used for Ethernet
#define TRANSFER_MEMORY_SIZE 0x00004000 // Size of the internal 16k USB SRAM

static U32 _aPool[ALLOC_SIZE / 4];      // Memory area used by the stack.

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory(_aPool, sizeof(_aPool));
  USBH_AssignTransferMemory((void * )TRANSFER_MEMORY_BASE, TRANSFER_MEMORY_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                     );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    );
  BSP_USBH_Init();
  USBH_OHC_Add((void * )OHCI_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.4.4 Renesas (formerly NEC)

For accessing the OHCI controller it is neccessary to enable access to the OHCI controller via the PCI controller. The initialization can be found in the BSP.c that comes with the eval-package. Otherwise the file can be obtained by asking SEGGER.
Additionally the controller is not capable of handling low-speed devices such as mouse and keyboards. In order to use such device, a USB hub is necessary.

Configuration example:

```
#define ALLOC_SIZE           0x8000 // Size of memory dedicated to the stack in bytes
#define OHCI_BASE_ADDRESS    0x002E0000
#define TRANSFER_MEMORY_BASE 0x20080000 // Startaddress of the internal 8k USB SRAM
#define TRANSFER_MEMORY_SIZE 0x00002000 // Size of the internal 8k USB SRAM

static U32 _aPool[ALLOC_SIZE / 4];      // Memory area used by the stack.

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory(_aPool, sizeof(_aPool));
  USBH_AssignTransferMemory((void * )TRANSFER_MEMORY_BASE, TRANSFER_MEMORY_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target.
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                     );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    );
  BSP_USBH_Init();
  USBH_OHC_Add((void * )OHCI_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
```


12.4.5 Toshiba TMPA900

The Toshiba TMPA900 has a fully integrated OHCI-compliant host controller.
It supports all kind of USB transfer-types, but has limited access to memory.
The OHCI controller only has access to the so called USB-RAM. This means that a transfer memory has to be specified. Within this transfer memory all relevant transfer descriptors and transfer memories are built. Typically 8kBytes are available for USB RAM, this limits the maximum connected devices to approx. 1-2.
As it states in the reference manual of the TMPA900 the controller is not capable of handling low-speed devices directly connected to the USB port.
After port pins and clocks are initialized the TMPA900 contains a non-conform Register (HcBCR0) that holds the OHCI controller in suspend state, this needs to be disabled.

Configuration example:

```
#define ALLOC_SIZE           0x10000 // Size of mem dedicated to the stack in bytes
#define OHCI_BASE_ADDRESS    0xF4500000
#define TRANSFER_MEMORY_BASE 0xF8008000
#define TRANSFER_MEMORY_SIZE 0x2000

#define ALLOC_BASE   (((U32)&_aPool[0]) + 0x4000000) // Use the non cached SDRAM area
static U32 _aPool[((ALLOC_SIZE + 256) / 4)];    // Memory area shall 256 byte aligned

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((U32 *)((ALLOC_BASE + 0xff) & ~0xff), ALLOC_SIZE);
  USBH_AssignTransferMemory((U32 *)TRANSFER_MEMORY_BASE, TRANSFER_MEMORY_SIZE);
  //
  // Configure the root hub
  //
  USBH_ConfigRootHub(0, 1, 0);
  //
  // Configure the number of devices and endpoints
  //
  USBH_ConfigMaxUSBDevices(2);
  USBH_ConfigTransferBufferSize(128);
  USBH_ConfigMaxNumEndpoints(4, 1, 0);
  //
  // External hub support
  //
  USBH_ConfigSupportExternalHubs(0);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                    );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                   );
  BSP_USBH_Init();
  USBH_OHC_Add((void*)OHCI_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.5 ST STM32 Driver

The USB host driver ST STM32 105/107 series works flawlessly with USB full-speed devices and can handle up to 127 devices. It currently supports Control, Bulk and Interrupt transfers.
Overcurrent support is not implemented by the controller since the overcurrent pin is not handled by the controller.

Configuration example:

```
#define ALLOC_SIZE              0xA000 // Size of mem dedicated to the stack in bytes
#define STM32_OTG_BASE_ADDRESS 0x50000000UL

static U32 _aPool[((ALLOC_SIZE + 256) / 4)];   // Memory area used by the stack.
                                               // add additional 256 bytes in
                                               // order to have a 256 byte aligned
                                               // address


void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((void *)(((U32)(&_aPool[0]) + 0xff) & ~0xffuL), ALLOC_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                    );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    | USBH_MTYPE_HID
                    | USBH_MTYPE_MSD
                   );
  BSP_USBH_Init();
  USBH_STM32_Add((void*)STM32_OTG_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.6 ST STM32F2_FS Driver

The USB host driver ST STM32 207/407 series works flawlessly with USB full-speed devices and can handle up to 127 devices. It currently supports Control, Bulk and Interrupt transfers.
Overcurrent support is not implemented by the controller since the overcurrent pin is not handled by the controller.

Configuration example:

```
#define ALLOC_SIZE              0x3500 // Size of mem dedicated to the stack in bytes
#define STM32_OTG_BASE_ADDRESS 0x40040000UL

static U32 _aPool[((ALLOC_SIZE + 256) / 4)]; // Memory shall be 256 byte aligned


static void _InitUSBHw(void) {
  U32 v;

  RCC_AHB1ENR |= 0
                | (1 <<  2)  // GPIOCEN: IO port C clock enable
                | (1 <<  0)  // GPIOAEN: IO port A clock enable
                | (1 <<  7)  // GPIOHEN: IO port H clock enable
                ;
  RCC_AHB2ENR |= 0
              | (1 <<  7)  // OTGFSEN: Enable USB OTG FS clock enable
              ;
  //
  // Set PA10 (OTG_FS_ID) as alternate function
  //
  v            = GPIOA_MODER;
  v           &= ~(0x3uL << (2 * 10));
  v           |=  (0x2uL << (2 * 10));
  GPIOA_MODER = v;
  v            = GPIOA_AFRH;
  v           &= ~(0xFuL << (4 * 2));
  v           |=  (0xAuL << (4 * 2));
  GPIOA_AFRH  = v;
  //
  // Set PA11 (OTG_FS_DM) as alternate function
  //
  v            = GPIOA_MODER;
  v           &= ~(0x3uL << (2 * 11));
  v           |=  (0x2uL << (2 * 11));
  GPIOA_MODER = v;
  v            = GPIOA_AFRH;
  v           &= ~(0xFuL << (4 * 3));
  v           |=  (0xAuL << (4 * 3));
  GPIOA_AFRH  = v;
  //
  // Set PA12 (OTG_FS_DP) as alternate function
  //
  v            = GPIOA_MODER;
  v           &= ~(0x3uL << (2 * 12));
  v           |=  (0x2uL << (2 * 12));
  GPIOA_MODER = v;
  v            = GPIOA_AFRH;
  v           &= ~(0xFuL << (4 * 4));
  v           |=  (0xAuL << (4 * 4));
  GPIOA_AFRH  = v;
  v            = GPIOH_MODER;
  v           &= ~(0x3uL << (2 * 5));
  v           |=  (0x1uL << (2 * 5));
  //
  // Set PA12 (OTG_FS_DP) as alternate function
  //
  GPIOH_MODER = v;
  GPIOH_BSRR  = (0x10000uL << 5);    // Initially clear LEDs

}
```



```
void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((void *)(((U32)(&_aPool[0]) + 0xff) & ~0xff), ALLOC_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                     );
   USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    | USBH_MTYPE_HID
                    | USBH_MTYPE_MSD
                    );
  BSP_USBH_Init();
  USBH_STM32F2_FS_Add((void*)STM32_OTG_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.7 ST STM32F2_HS Driver

The USB host driver ST STM32 207/407 series works flawlessly with all USB speed classes (low, full and high speed) and can handle up to 127 devices. It currently supports Control, Bulk and Interrupt transfers.

Configuration example:

```
#define ALLOC_SIZE              0xA000 // Size of mem dedicated to the stack in bytes
#define STM32_OTG_BASE_ADDRESS 0x50000000UL

static U32 _aPool[((ALLOC_SIZE + 256) / 4)]; // Memory shall be 256 byte aligned

static void _InitUSBHw(void) {
  RCC_AHB1ENR |= 0
              | (1 <<  8)  // GPIOIEN: IO port I clock enable
              | (1 <<  7)  // GPIOHEN: IO port H clock enable
              | (1 <<  2)  // GPIOCEN: IO port C clock enable
              | (1 <<  1)  // GPIOBEN: IO port B clock enable
              | (1 <<  0)  // GPIOAEN: IO port A clock enable
              ;
  //
  // UPLI data pins
  // PA3 (OTG_HS_ULPI alternate function, DATA0)
  //
  GPIOA_MODER    =   (GPIOA_MODER  & ~(3UL  <<  6)) | (2UL  <<  6);
  GPIOA_OTYPER  &=  ~(1UL  <<  3);
  GPIOA_OSPEEDR |=   (3UL  <<  6);
  GPIOA_PUPDR   &=  ~(3UL  <<  6);
  GPIOA_AFRL     =   (GPIOA_AFRL  & ~(15UL << 12)) | (10UL << 12);
  //
  //PB0, PB1 (OTG_HS_ULPI alternate function, DATA1, DATA2)
  //
  GPIOB_MODER    =   (GPIOB_MODER  & ~(15UL <<  0)) | (10UL <<  0);
  GPIOB_OTYPER  &=  ~(3UL  <<  0);
  GPIOB_OSPEEDR |=   (15UL <<  0);
  GPIOB_PUPDR   &=  ~(15UL <<  0);
  GPIOB_AFRL     =   (GPIOB_AFRL  & ~(0xFFUL <<  0)) | (0xAA <<  0);
  //
  // PB10..13 (OTG_HS_ULPI alternate function, DATA3 to DATA6)
  //
  GPIOB_MODER    =   (GPIOB_MODER  & ~(0xFFUL << 20)) | (0xAA << 20);
  GPIOB_OTYPER  &=  ~(15UL << 10);
  GPIOB_OSPEEDR |=   (0xFFUL << 20);
  GPIOB_PUPDR   &=  ~(0xFFUL << 20);
  GPIOB_AFRH     =   (GPIOB_AFRH  & ~(0xFFFFUL << 8)) | (0xAAAA << 8);
  //
  // PB5 (OTG_HS_ULPI alternate function, DATA7)
  //
  GPIOB_MODER    =   (GPIOB_MODER  & ~(3UL  <<  10)) | (2UL  <<  10);
  GPIOB_OTYPER  &=  ~(1UL  <<  5);
  GPIOB_OSPEEDR |=   (3UL  << 10);
  GPIOB_PUPDR   &=  ~(3UL  << 10);
  GPIOB_AFRL     =   (GPIOB_AFRL  & ~(15UL <<  20)) | (10UL <<  20);
  //
  // ULPI control pins
  // PC0 (OTG_HS_ULPI alternate function, STP)
  //
  GPIOC_MODER    =   (GPIOC_MODER & ~(3UL  <<   0)) | (2UL  <<  0);
  GPIOC_OSPEEDR |=   (3UL  <<  0);
  GPIOC_AFRL     =   (GPIOC_AFRL  & ~(15UL <<   0)) | (10UL <<  0);
  //
  // PI11 (OTG_HS_ULPI alternate functon, DIR)
  //
  GPIOI_MODER    =   (GPIOI_MODER & ~(3UL  <<  22)) | (2UL  << 22);
  GPIOI_OSPEEDR |=   (3UL  << 22);
  GPIOI_AFRH     =   (GPIOI_AFRH  & ~(15UL <<  12)) | (10UL << 12);
  //
  // PH4 (OTG_HS_ULPI alternate function, NXT)
  //
  GPIOH_MODER    =   (GPIOH_MODER & ~(3UL  <<   8)) | (2UL  <<  8);
  GPIOH_OSPEEDR |=   (3UL  <<  8);
  GPIOH_AFRL     =   (GPIOH_AFRL  & ~(15UL <<  16)) | (10UL << 16);
  //
  // PA5 (OTG_HS_ULPI a;ternate function, CLOCK)
  //
  GPIOA_MODER    =   (GPIOA_MODER & ~(3UL  <<  10)) | (2UL  << 10);
  GPIOA_OSPEEDR |=   (3UL  << 10);
  GPIOA_AFRL     =   (GPIOA_AFRL  & ~(15UL <<  20)) | (10UL << 20);
```



```
  //
  //  Enable clock for OTG_HS and OTGHS_ULPI
  //
  RCC_AHB1ENR     |=  (3UL << 29);
  USBH_OS_Delay(100);
  //
  // Reset OTGHS clock
  //
  RCC_AHB1RSTR    |=  (1UL << 29);
  USBH_OS_Delay(100);
  RCC_AHB1RSTR   &= ~(1UL << 29);
  USBH_OS_Delay(400);
}

/*********************************************************************
*
*       _ISR
*
*  Function description
*/
static void _ISR(void) {
  USBH_ServiceISR(0);
}

/*********************************************************************
*
*       USBH_X_Config
*
*  Function description
*/
void USBH_X_Config(void) {
  USBH_AssignMemory((U32 *)(((U32)(&_aPool[0]) + 0xff) & ~0xffuL), ALLOC_SIZE);   //
Assigning memory should be the first thing
//  USBH_AssignTransferMemory((void*)TRANSFER_MEMORY_BASE, TRANSFER_MEMORY_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0xFFFFFFFF); // 0xFFFFFFFF: Do not filter: Output all warnings.
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    | USBH_MTYPE_HID
                   );
  _InitUSBHw();
  USBH_STM32F2_HS_Add((void*)STM32_OTG_BASE_ADDRESS);
  BSP_USB_InstallISR_Ex(USB_ISR_ID, _ISR, USB_ISR_PRIO);
}
```


12.8 Renesas RX Driver

The RX62 host driver fully supports all kind devices connected to the USB host controller's port, which means low-speed, full-speed and high-speed (which operate at full-speed) devices work flawlessly. Any kind of transfer-type (Control,Bulk,Interrupt and Isochronous) is supported.
The limitation of the controller is that 5 devices can simultaneously be connected.

Configuration example:

```
#define ALLOC_SIZE            0x3000 // Size of mem dedicated to the stack in bytes
#define USB0_BASE_ADDRESS     0x000A0000UL

static U32 _aPool[((ALLOC_SIZE + 256) / 4)]; // Memory shall be 256 byte aligned

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((void *)(((U32)(&_aPool[0]) + 0xff) & ~0xff), ALLOC_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  USBH_RX62_Add((void*)USB0_BASE_ADDRESS);
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                     );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    );
  BSP_USBH_Init();
  USBH_RX62_Add((void*)STM32_OTG_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.9 Atmel AVR32 Driver

The AVR32 host driver fully supports all kind devices connected to the USB host controller's port, which means low-speed, full-speed and high-speed (which operate at full-speed) devices work flawlessly. Any kind of transfer-type (Control,Bulk,Interrupt and Isochronous) is supported.
The limitation of the controller is that pipes have to be reserved for a specific device after the device has been successfully enumerated.

For the Atmel AVR32 driver the EP_RAM location needs to be configured. This can be done by using the `USBH_AVR32_ConfigureEPRAM()` function.

Configuration example:

```
#define ALLOC_SIZE           0x2000 // Size of memory dedicated to the stack in bytes
#define USBB_BASE_ADDRESS     0xFFFE0000UL
#define USBB_RAM_BASE_ADDRESS 0xE0000000UL

static U32 _aPool[((ALLOC_SIZE + 256) / 4)];   // Memory area used by the stack.
                                               // add additional 256 bytes in order
                                               // to have a 256 byte aligned address

void USBH_X_Config(void) {
 //
 // Assigning memory should be the first thing
 //
  USBH_AssignMemory((void *)(((U32)(&_aPool[0]) + 0xff) & ~0xffUL), ALLOC_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0
                     | USBH_MTYPE_INIT
                     | USBH_MTYPE_HID
                     | USBH_MTYPE_MSD
                     | USBH_MTYPE_APPLICATION
                     );
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    );
  BSP_USBH_Init();
  USBH_AVR32_Add((void*)USBB_BASE_ADDRESS);
  USBH_AVR32_ConfigureEPRAM(USBB_RAM_BASE_ADDRESS);
  BSP_USBH_InstallISR(_ISR);
}
```


12.10 Freescale Kinetis FullSpeed Driver

The Freescale Kinetis host driver fully supports all kind devices connected to the USB host controller's port, which means low-speed, full-speed and high-speed (which operate at full-speed) devices work flawlessly. Any kind of transfer-type (Control, Bulk, Interrupt and Isochronous) is supported.
The limitation of the controller is that only one device can be connected.

Configuration example:

```
#define ALLOC_SIZE          0xC000  // Size of memory dedicated to the stack in bytes
#define USB_OTG_BASE_ADDR   0x40072000

static U32 _aPool[((ALLOC_SIZE + 256) / 4)];    // Memory area used by the stack.
                                                // add additional 256 bytes in order
                                                // to have a 256 byte aligned address

void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory((void *)(((U32)(&_aPool[0]) + 0xff) & ~0xffuL), ALLOC_SIZE);
  //
  // Define log and warn filter
  // Note: The terminal I/O emulation affects the timing
  // of your communication, since the debugger stops the target
  // for every terminal I/O output unless you use DCC!
  //
  USBH_SetWarnFilter(0xFFFFFFFF);  // 0xFFFFFFFF: Do not filter: Output all warnings.
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    | USBH_MTYPE_HID
                    );
  _InitUSBHw();
  USBH_KINETIS_FS_Add((void*)(USB_OTG_BASE_ADDR));
  BSP_USB_InstallISR_Ex(USB_ISR_ID, _ISR, USB_ISR_PRIO);
}
```


12.11 EHCI Driver

This driver can handle all USB transfers - this means it can handle Control, Interrupt, Bulk and Isochronous transfers. It handles up to 127 devices and has an automatic error detection.

12.11.1 General information

For MCUs with internal data cache controller such as ARM9/ARM11/Cortex A8 etc., please note the following:
In order to avoid cache inconsistency, the allocation pool for emUSB-Host should be in a non-cached,non-buffered RAM region.

Chapter 13

USB On The Go (Add-On)

This chapter describes the emUSB-Host add-on emUSB-OTG and how to use it. The emUSB-OTG is an optional extension of emUSB-Host.

13.1 Introduction

13.1.1 Overview

USB On-The-Go (OTG) allows two USB devices to “talk” to each other.
OTG introduces the dual-role device, meaning a device capable of functioning as either host or peripheral.
USB OTG retains the standard USB host/peripheral model, in which a single host talks to USB peripherals.
emUSB OTG offers a simple interface in order to detect the role of the USB OTG controller.

13.1.2 Features

The following features are provided:

- Detection of the USB role of the device.
- Virtually any USB OTG transceiver can be used.
- Simple interface to OTG-hardware.
- Seamless integration with emUSB Host and emUSB Device.

13.1.3 Example code

An example application which uses the API is provided in the USB_OTG_Start.c file of your shipment. This example starts the OTG stack and waits until a valid session is detected. As soon as a valid session is detected, the ID-pin state is checked to detect whether emUSB Device or emUSB Host shall then be initialized. For emUSB Device a simple mouse sample is used. On emUSB host side an MSD-sample is used that detects USB memory stick and shows information about the detected stick.

Excerpt from the example code:

```
/*********************************************************************
*
*       OTGTask
*
* Function description
*   USB OTG handling task.
*   It implements a basic function how to check which USB stack shall be called.
*   It first checks whether the OTG chip has detected a valid session.
*   If so, the next step will be to check the state of the ID-pin of the cable.
*   If pin is 0 (grounded) -> a USB host cable is connected.
*   If pin is 1 (floating) -> a USB device is plugged in.
*
*/
void OTGTask(void);
void OTGTask(void) {
  int State;
  while (1) {
    //
    // Initialize OTG stack
    //
    USB_OTG_Init();
    //
    // Wait for a valid session
    //
```



```
    while (1) {
      if (USB_OTG_IsSessionValid()) {
        break;
      }
      USB_OTG_OS_Delay(50);
      BSP_ToggleLED(0);
    }
    //
    // Determine whether Device or Host stack shall be initialized and started.
    //
    State = USB_OTG_GetIdState();
    USB_OTG_DeInit();
    USB_OS_Delay(10);
    if (State == USB_OTG_ID_PIN_STATE_IS_HOST) {
      _ExecUSBHost();
    } else if (State == USB_OTG_ID_PIN_STATE_IS_DEVICE) {
      _ExecUSBDevice();
    }
  }
}
```


13.2 Driver

13.2.1 General information

To use emUSB OTG, a driver matching the target hardware is required. The code size of a driver depends on the hardware and is typically between 1 and 3 Kbytes. The driver handles both the OTG controller as well as the OTG transceiver.

The driver interface has been designed to allow support of internal and external OTG controllers. It also allows to take full advantage of hardware features such as session detection and session request protocol.

13.2.2 Available drivers

emUSB OTG drivers are optional components to emUSB OTG. The following drivers are available:

Device	Identifier
Renesas RX62	USB_OTG_Driver_Renesas_RX62N

To add a driver to emUSB OTG, USB_OTG_AddDriver() should be called with the proper identifier before emUSB-OTG starts any session detection. Refer to USB_OTG_AddDriver() for detailed information.

13.2.2.1 Renesas RX62

This driver supports all Renesas RX62 MCU with integrated USB OTG controllers. The current USB-OTG transceiver that is supported with this driver is:
Analogic TECH AAT3125

Configuring the driver:

To add the driver, use USB_OTG_AddDriver() with the driver identifier USB_OTG_DRIVER_Renesas_RX62N. This function has to be called from USB_OTG_X_Config(). Refer to USB_OTG_AddDriver(). and IP_X_Configure() for more information.

Example

```
void USB_OTG_X_Config(void) {
  USB_OTG_AddDriver(&USB_OTG_Driver_Renesas_RX62N);     // Add a driver to USB OTG.
}
```


13.3 API Functions

This chapter describes the emUSB-OTG API functions. These functions are defined in the header file USB_OTG.h.

Function	Description
USB_OTG_Init()	Initializes the emUSB-OTG stack.
USB_OTG_DeInit()	Deinitializes emUSB-OTG stack.
USB_OTG_GetIdState()	Returns the state of the ID-pin.
USB_OTG_GetVBUSState()	Returns the VBus voltage.
USB_OTG_IsSessionValid()	Returns whether the detected OTG-state is valid.
USB_OTG_AddDriver()	Adds a OTG-driver to the stack.
USB_OTG_X_Config()	User-provided function which configures the emUSB-OTG stack.

Table 13.1: emUSB OTG API function overview

13.3.1 USB_OTG_Init()

Description

Initializes the USB OTG core.

Prototype

```
void USB_OTG_Init(void);
```

Additional Information

The function will initally call the OS-Layer initialization, then the user-provided USB_OTG_X_Config() and will then call the initialization routine of the driver.

13.3.2 USB_OTG_DeInit()

Description

Deinitialize emUSB-OTG stack.

Prototype

```
void USB_OTG_DeInit(void);
```

Additional Information

It will deinitialize the complete OTG module. It removes/releases all OS-layer relevant resources and calls the driver deinitialization callback.

13.3.3 USB_OTG_GetIdState()

Description

Returns the current state of the ID-pin.

Prototype

```
int USB_OTG_GetIdState(void);
```

Return Value

`USB_OTG_ID_PIN_STATE_IS_HOST:`	OTG DEVICE shall be used as host.
`USB_OTG_ID_PIN_STATE_IS_DEVICE:`	OTG DEVICE shall be used as device.

Additional information

In order to select the correct session (Host or Device), a OTG transceiver should detect a valid session. The ID-States are defined as follows:

- ID-pin is 0 (grounded) - a USB host cable is connected.
- ID-pin is 1 (floating) - a USB device is plugged in.

13.3.4 USB_OTG_GetVBUSState()

Description

Returns the current state of the VBUS via an OTG transceiver.

Prototype

```
int USB_OTG_GetVBUSState(void);
```

Return Value

Returns the voltage given in mVolt.

Additional information

This function can be used to check the voltage that is measured on the VBUS-line of the USB-Bus.

13.3.5 USB_OTG_IsSessionValid()

Description

Returns whether the OTG transceiver has marked the session as valid.

Prototype

```
int USB_OTG_IsSessionValid(void);
```

Return Value

`== 0:` Session is not valid.
`== 1:` Session is valid.

Additional information

Before any decision can be made by emUSB-OTG, the USB OTG controller or the OTG-transceiver must detect a valid session. If this is not the case, it is most likely that there is no cable conntected to the USB OTG-port.

13.3.6 USB_OTG_AddDriver()

Description

Adds a OTG-driver to the stack.

Prototype

```
void USB_OTG_AddDriver(const USB_OTG_HW_DRIVER * pDriver);
```

Parameter

Parameter	Description
pDriver	[IN] - Pointer to the driver structure.

Table 13.2: USBH_PRINTER_GetPortStatus() parameter list

Additional information

Adds a OTG driver to the OTG stack. This function is generally called in the USB_OTG_X_AddDriver.
Refer to *Available drivers* on page 266 for a list of available OTG drivers.

13.3.7 USB_OTG_X_Config()

Description

Helper function to prepare and configure the USB OTG stack.

Prototype

```
void USB_OTG_X_Config(void);
```

Additional information

This function is called by the startup code of the USB OTG stack from `USB_OTG_Init()`. Refer to *USB_OTG_Init()* on page 268 for more information.

Chapter 14

Debugging

emUSB-Host comes with various debugging options. These include optional warning and log outputs, as well as other runtime options which perform checks at run time.

14.1 Message output

The debug builds of emUSB-Host include a fine-grained debug system which helps to analyze the correct implementation of the stack in your application. All modules of the emUSB-Host stack can output logging and warning messages via terminal I/O, if the specific message type identifier is added to the log and/or warn filter mask. This approach provides the opportunity to get and interpret only the logging and warning messages which are relevant for the part of the stack that you want to debug.

By default, all warning messages are activated in all emUSB-Host sample configuration files. All logging messages are disabled except for the messages from the initialization phase.

14.2 API functions

Function	Description
USBH_AddLogFilter()	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
USBH_AddWarnFilter()	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
USBH_Log()	Called if emUSB-Host wants to log a message.
USBH_Panic()	Called if emUSB-Host encounters a critical situation.
USBH_SetLogFilter()	Sets the mask that defines which logging message should be displayed.
USBH_SetWarnFilter()	Sets the mask that defines which warning message should be displayed.
USBH_Warn()	Called if emUSB-Host generates a warning message.

Table 14.1: emUSB-Host debugging API function overview

14.2.1 USBH_AddLogFilter()

Description

Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.

Prototype

```
void USBH_AddLogFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which logging messages should be added to the filter mask.

Table 14.2: USBH_AddLogFilter() parameter list

Additional information

This function can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

For available message types See “Message types” on page 285.

14.2.2 USBH_AddWarnFilter()

Description

Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.

Prototype

```
void USBH_AddWarnFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which warning messages should be added to the filter mask.

Table 14.3: USBH_AddWarnFilter() parameter list

Additional information

This function can also be used to remove a filter condition which was set before. It adds/removes the specified filter to/from the filter mask via a disjunction.

For available message types See “Message types” on page 285.

14.2.3 USBH_Log()

Description

Called by emUSB-Host in debug builds to log a message.

Prototype

```
void USBH_Log(const char * s);
```

Parameter

Parameter	Description
s	Pointer to the string to log.

Table 14.4: USBH_Log() parameter list

Additional information

In a release build this function is not linked in.

Example

See the implementation found in the `USBH_ConfigIO.c` file of your shipment.

14.2.4 USBH_Panic()

Description

Called by emUSB-Host in debug builds if a critical situation is encountered.

Prototype

```
void USBH_Panic(const char * sError);
```

Parameter

Parameter	Description
sError	Pointer to the error string.

Table 14.5: USBH_Log() parameter list

Additional information

In a release build this function is not linked in. The default implementation of this function disables all interrupts to avoid further task switches, outputs sError via terminal I/O and loops forever. When using an emulator, you should set a break-point at the beginning of this routine or simply stop the program after a failure.

Example

See the implementation found in the `USBH_ConfigIO.c` file of your shipment.

14.2.5 USBH_SetLogFilter()

Description

Sets a mask that defines which logging message should be logged. Logging messages are only available in debug builds of emUSB-Host.

Prototype

```
void USBH_SetLogFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which logging messages should be displayed.

Table 14.6: USBH_SetLogFilter() parameter list

Additional information

Should be called from USBH_X_Config(). By default, the filter condition USBH_MTYPE_INIT is set.

Please note that the more logging is enabled, the more the timing of the application is influenced.

For available message types See “Message types” on page 285.

14.2.6 USBH_SetWarnFilter()

Description

Sets a mask that defines which warning messages should be logged. Warning messages are only available in debug builds of emUSB-Host.

Prototype

```
void USBH_SetWarnFilter(U32 FilterMask);
```

Parameter

Parameter	Description
FilterMask	Specifies which warning messages should be displayed.

Table 14.7: USBH_SetWarnFilter() parameter list

Additional information

Should be called from USBH_X_Config(). By default, all filter conditions are set.

For available message types See “Message types” on page 285.

14.2.7 USBH_Warn()

Description

Called by emUSB-Host in debug builds to log an error message.

Prototype

```
void USBH_Warn(const char * s);
```

Parameter

Parameter	Description
s	Pointer to the string to log.

Table 14.8: USBH_Warn() parameter list

Additional information

In a release build this function is not linked in.

Example

See the implementation found in the `USBH_ConfigIO.c` file of your shipment.

14.3 Message types

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings.

Symbolic name	Description
USBH_MTYPE_APPLICATION	Activates the output of messages from the application.
USBH_MTYPE_CORE	Activates the output of messages from the core of the stack.
USBH_MTYPE_DEVICE	Activates the output of messages from the device handling logic.
USBH_MTYPE_DRIVER	Activates the output of messages from the hardware driver.
USBH_MTYPE_HID	Activates the output of messages from HID component.
USBH_MTYPE_HUB	Activates the output of messages from the hub handling logic.
USBH_MTYPE_INIT	Activates the output of messages from the initialization of the stack.
USBH_MTYPE_MEM	Activates the output of messages from the memory management module.
USBH_MTYPE_MSD	Activates the output of messages from the MSD component.
USBH_MTYPE_OHCI	Activates the output of messages from the Open Host Controller Interface.
USBH_MTYPE_PNP	Activates the output of messages from the enumeration process.
USBH_MTYPE_UBD	Activates the output of messages from the USB bus driver.
USBH_MTYPE_EHCI	Activates the output of messages from the Enhanced Host Controller Interface.
USBH_MTYPE_TIMER	Activates the output of messages from the timer related routines.
USBH_MTYPE_URB	Activates the output of messages from the USB Request Block related routines.
USBH_MTYPE_EP	Activates the output of messages from the endpoint related routines.
USBH_MTYPE_PRINTER_CLASS	Activates the output of messages from the printer module.
USBH_MTYPE_CDC	Activates the output of messages from the communication device class module.
USBH_MTYPE_FT232	Activates the output of messages from the FT232 module.

Table 14.9: emUSB-Host message types

Chapter 15

OS integration

emUSB-Host is designed to be used in a multitasking environment. The interface to the operating system is encapsulated in a single file, the USB-Host/OS interface. For emUSB-Host, all functions required for this USB-Host/OS interface are implemented in a single file which comes with emUSB-Host.

This chapter provides descriptions of the functions required to fully support emUSB-Host in multitasking environments.

15.1 General information

All OS interface functions for emUSB-Host are implemented in `USBH_OS_embOS.c` which is located in the USBH folder of your shipment.

15.2 OS layer API functions

Function	Description
	General macros
USBH_OS_Delay()	Blocks the calling task for a given time.
USBH_OS_DisableInterrupt()	Disables interrupts.
USBH_OS_EnableInterrupt()	Enables interrupts.
USBH_OS_GetTime32()	Returns the current system time in ticks with a resolution of one ms. On 32-bit systems, the value will wrap around after approximately 49.7 days. This is taken into account by the stack.
USBH_OS_Init()	Creates and initializes all objects required for task synchronization. These are 2 events (for USBH_Task() and USBH_ISRTask()) and one semaphore for protection of critical code which may not be executed from multiple task at the same time.
USBH_OS_Lock()	The stack requires a single lock, typically a resource semaphore or mutex. This function locks this object, guarding sections of the stack code against other tasks. If the entire stack executes from a single task, no functionality is required here.
USBH_OS_Unlock()	Unlocks the single lock used locked by a previous call to USBH_OS_Lock().
	USBH_Task synchronization
USBH_OS_SignalNetEvent()	Wakes the USBH_Task() if it is waiting for a NET-event or timeout in the function USBH_OS_WaitNetEvent().
USBH_OS_WaitNetEvent()	Called from USBH_Task() only. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalNetEvent() is called from an other task or ISR.
	USBH_ISRTask synchronization
USBH_OS_SignalRxEvent()	Wakes the USBH_ISRTask() if it is waiting for a NET-event or timeout in the function USBH_OS_WaitRxEvent().
USBH_OS_WaitRxEvent()	Optional. Called from USBH_ISRTask(), if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalRxEvent() is called from the ISR.
	Application task synchronization
USBH_OS_WaitItem()	Suspends a task which needs to wait for an object. This object is identified by a pointer to it and can be of any type, for example a socket.
USBH_OS_WaitItemTimed()	Suspends a task which needs to wait for an object. This object is identified by a pointer to it and can be of any type, for example a socket. The second parameter defines the maximum time in timer ticks until the event has to be signaled.
USBH_OS_SignalItem()	Sets an event object to signaled state, or resumes tasks which are waiting at the event object. Function is called from a task, not an ISR.

Table 15.1: Target OS interface function list

Chapter 16

Performance & resource usage

This chapter covers the performance and resource usage of emUSB-Host. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

16.1 Memory footprint

emUSB-Host is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. Note that the values are only valid for the given configuration.

The tests were run on a 32-bit CPU running at 72MHz. The test program was compiled with size optimization.

16.1.1 ROM

The following table shows the ROM requirement of emUSB-Host:

Description	ROM
emUSB-Host core incl. driver	app. 20 KBytes
HID class support	app. 5 KBytes
MSD class support	app. 8 KBytes + sizeof(Filesystem)*

* ROM size of emFile File system is app. 10KBytes

The ROM requirement of an interface driver is about 1.5 Kbytes.

16.1.2 RAM

The following table shows the RAM requirement of emUSB-Host:

Description	RAM
emUSB-Host core incl. driver	app. 20 Kbytes

The RAM requirement of an interface driver is about 1.5 Kbytes.

16.2 Performance

The tests were run on a 32-bit CPU running at 72MHz with a full-speed host controller (OHCI). The device used for testing was a J-Link.

The following table shows the send and receive speed of emUSB-Host:

Description	Speed
Bulk	
Send speed	800 KByte/sec
Receive speed	760 KByte/sec

Chapter 17

Related Documents

- Universal Serial Bus Specification 1.1, http://www.usb.org
- Universal Serial Bus Specification 2.0, http://www.usb.org
- USB device class specifications (Audio, HID, Printer, etc.), http://www.usb.org
- USB 2.0, Hrsg. H. Kelm, Franzi’s Verlag, 2001, ISBN 3-7723-7965-6

Chapter 18

Glossary

CPU	Central Processing Unit. The "brain" of a microcontroller; the part of a processor that carries out instructions.
EOT	End Of Transmission.
FIFO	First-In, First-Out.
ISR	Interrupt Service Routine. The routine is called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
RTOS	Real-time Operating System.
Scheduler	The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.
Stack	An area of memory with LIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.
Task	A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.
Tick	The OS timer interrupt. Usually equals 1 ms.

Chapter 19

FAQ

This chapter answers some frequently asked questions.

Q: Which CPUs can I use emUSB with?
A: It can be used with any CPU (or MPU) for which a C compiler exists. Of course, it will work faster on 16/32-bit CPUs than on 8-bit CPUs.

Q: Do I need a real-time operating system (RTOS) to use the emUSB-Host stack?
A: Yes, an RTOS is required.

Q: Is the emUSB-Host API thread-safe?
A: No. The user is responsible for locking the API calls against each other.

Q: emUSB-Host does not compile because of missing includes in the USBH_MSD_FS.c file.
A: The USBH_MSD_FS.c file is a file system layer for emUSB-Host's MSD module. This layer is written for the SEGGER file system emFile.
If you do not have emFile you can use this file as a sample to writte a file system layer for your own file system.

Q: Devices connected directly to my Host work, but do not work when connected through a hub.
A: Please check your USBH_X_Config function. In it you should call USBH_ConfigSupportExternalHubs(1) to enable hub support.

Q: When I enable all logs via USBH_SetLogFilter(0xffffffff) my devices no longer enumerate.
A: Enabling too many log outputs can drastically influence the timing of the application up to a point where it may no longer function. It is best practice to limit the number of logs only to the ones you are interested in.

Index

