# emUSB-Host

## CPU independent USB Host stack for embedded applications

## User Guide & Reference Manual

Document: UM10001
Software Version: 2.12
Revision: 0
Date: Juli 9, 2018

A product of SEGGER Microcontroller GmbH

www.segger.com

## Disclaimer



## Copyright notice



## Trademarks



## Contact address

SEGGER Microcontroller GmbH

In den Weiden 11
D-40721 Hilden

Germany

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

## Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: Juli 9, 2018

| Software | Revision | Date | By | Description |
|---|---|---|---|---|
| 2.12 | 0 | 180709 | YR | Update to latest software version. |
| 2.10 | 0 | 180618 | YR | Added LAN chapter.<br>Chapter “USB Host Core”:<br>• Added function USBH_SetRootPortPower()<br>• Added function USBH_HUB_SuspendResume()<br>Chapter “CDC”:<br>• Added function USBH_CDC_SuspendResume().<br>Chapter “Bulk”:<br>• Added function USBH_BULK_SetupRequest().<br>Added section *LPC54xxx High Speed driver*. |
| 2.08a | 0 | 180518 | RH | Update to latest software version. |
| 2.08 | 0 | 180515 | RH | Added section *Updating emUSB-Host*.<br>Added section *ATSAMx7 driver*. |
| 2.07 | 0 | 180220 | RH | Chapter “HID”:<br>• Added function USBH_HID_SetOnGenericEvent(). |
| 2.06a | 0 | 180118 | RH | Chapter “HID”:<br>• Function USBH_HID_GetReportDescriptor() replaced by new function USBH_HID_GetReportDesc().<br>• Update description for USBH_HID_GetReport().<br>• Added “DeviceType” to USBH_HID_DEVICE_INFO. |
| 2.06 | 0 | 180108 | RH | Section “Synopsys DWC2 driver”<br>• Added STM32H7 driver specific functions.<br>Chapter “HID”:<br>• Added function USBH_HID_SetIndicators().<br>• Added function USBH_HID_GetIndicators().<br>• Added function USBH_HID_SetReportEx(). |
| 2.04 | 0 | 171208 | RH | Added vendor (BULK) class driver. |
| 2.02 | 0 | 171130 | RH | Update RAM usage values. |
| 2.00b | 0 | 171016 | YR | Update to latest software version. |
| 2.00a | 0 | 170921 | YR | Update to latest software version.<br>Updated Performance & resource usage chapter. |
| 2.00 | 0 | 170915 | RH | Major revision of the manual.<br>• Manual converted to text processor emDoc.<br>• Chapter “Running emUSB-Host on target hardware” revised.<br>• Chapter “Configuring emUSB-Host” revised.<br>• All API function descriptions synchronized with source code. |
| 1.30e | 0 | 170717 | YR | Update to latest software version. |
| 1.30d | 0 | 170610 | RH | Removed obsolete USBH_FUNCTION_SET_CONFIGURATION. |
| 1.30c | 0 | 170517 | YR | Update to latest software version. |
| 1.30b | 0 | 170425 | RH | Update to latest software version. |
| 1.30a | 0 | 170306 | RH | Update to latest software version. |
| 1.30 | 0 | 170301 | YR | Chapter “CDC”:<br>• Updated _USBH_CDC_DEVICE_INFO description.<br>Chapter “HID”:<br>• Added USBH_HID_RW_CONTEXT description. |
| 1.20c | 0 | 160928 | YR | Chapter “USB Host Core”:<br>• Added USBH_SetOnSetPortPower() |
| internal | 0 | 160915 | YR | Chapter “MTP”:<br>• Added USBH_MTP_ConfigEventSupport()<br>• Added USBH_MTP_GetEventSupport()<br>• Added USBH_MTP_GetObjectPropsSupported() |
| 1.20b | 2 | 160609 | YR | Chapter “MTP”: |

| Software | Revision | Date | By | Description |
|---|---|---|---|---|
| | | | | • Added `USBH_MTP_CheckLock()` |
| 1.16f | 0 | 160422 | YR | Update to latest software version. |
| internal | 0 | 160215 | YR | Chapter “MTP”:<br>• Added `USBH_MTP_SetEventCallback()` |
| internal | 0 | 151215 | YR | Updated chapter “MTP device driver”.<br>• Updated description of the `USBH_SubmitUrb()` function. |
| internal | 0 | 151011 | YR | Added new chapter “MTP device driver”. |
| 1.16e | 0 | 150729 | SR | Update to latest software version. |
| 1.16d | 0 | 150713 | YR | Chapter “Performance & resource usage”:<br>• Updated with detailed values for each USB class.<br>• Several small improvements. |
| 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”:<br>• Added `USBH_EHCI_Config_SetM2MEndianMode()` |
| 1.16 | 0 | 141208 | YR | Chapter “CDC”:<br>• Added `USBH_CDC_SetConfigFlags()` |
| internal | 0 | 141117 | YR | Several improvements to descriptions.<br>Minor spelling corrections. |
| internal | 0 | 140916 | YR | Several improvements to descriptions. |
| 1.15b | 0 | 140829 | YR | Chapter “CDC”:<br>• Added `USBH_CDC_ReadAsync()`<br>• Added `USBH_CDC_WriteAsync()`<br>• Added `USBH_CDC_RW_CONTEXT`<br>• 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:<br>• Added new drivers to the list.<br>• Updated performance values. |
| 1.10 | 0 | 120515 | YR | Chapter “FT232” and chapter “CDC” added.<br>Chapter Host controller specifics:<br>• Added new drivers to the list. |
| 1.08 | 2 | 111114 | SR | Added new driver for Atmel AVR32.<br>Updated cross-references.<br>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”<br>Added pictures to chapter HID, MSD, Printer |

| Software | Revision | Date | By | Description |
|---|---|---|---|---|
| | | | | Updated Configuration chapter<br>Added Sample app chapter<br>Added information that a RTOS is necessary<br>Updated Information in chapter Introduction<br>Update functions descriptions in chapter API.<br>Added new driver configuration in chapter Configuration |
| 1.02 | 0 | 100806 | MD | Added screenshots to chapter “Running emUSB-Host on target hardware”.<br>Renamed function parameters to conform with our coding standards.<br>Changed the return values of HID API functions to USBH_STATUS.<br>Added detail descriptions to example applications. |
| 1.01 | 0 | 100721 | MD | Chapter “Printer” added.<br>Corrected various function prototypes. |
| 1.00 | 0 | 090609 | AS | Initial version. |

# 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 the product 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 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 emUSB-Host 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 separate task(s) beside the target application task. This implicitly 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

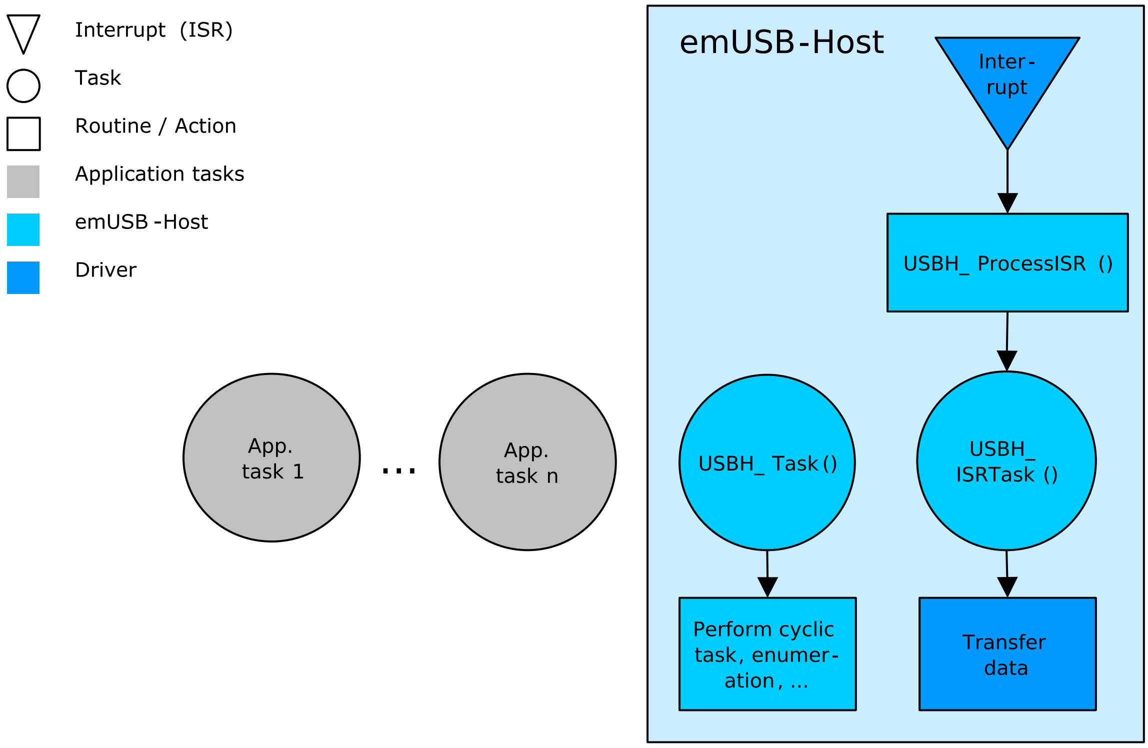
emUSB-Host uses two dedicated tasks. One of the tasks processes the interrupts generated by the USB host controller. The function `USBH_ISRTask()` should run as this task with the highest priority. The other task manages the internal software timers. Its routine should be the `USBH_Task()` function.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

# USB Background information

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

## 2.1 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.

## 2.2 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 Mbit/s for low speed and 12 Mbit/s 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 Mbit/s (40 times faster than USB 1.1 at full-speed) was added.

### USB 3.0 (November 2008)

As all previous USB standards, USB 3.0 is fully forward and backward compatible. Existing cables and connectors may be reused but the new speed can only be used with new USB 3.0 cables and devices. The new speed class is named USB Super-Speed, which offers a maximum rate of 5 Gbit/s.

### USB 3.1 (July 2013)

As all previous USB standards, USB 3.1 is fully forward and backward compatible. The new specification replaces the 3.0 standard and introduces new transfer speeds of up to 10 Gbit/s.

# 2.3 USB System Architecture

An 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

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

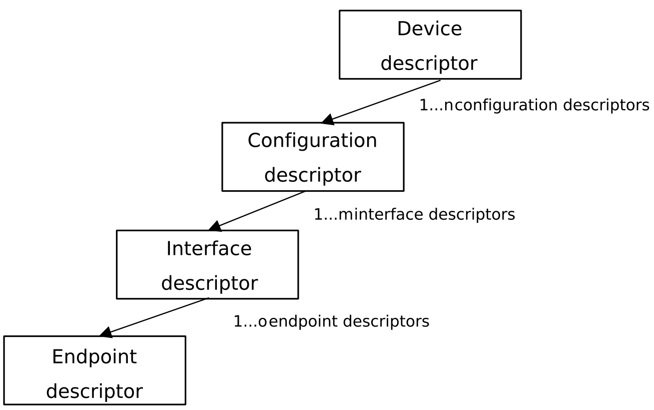
- A driver for the USB host controller hardware.
- The USB host stack which implements the high level functions used by the USB class drivers (including enumeration and hub support).
- One or more USB class drivers providing generic access to certain types of USB devices such as printers or mass storage devices.

### USB Device

Two types of devices exist: Hubs and functions. Hubs usually provide four ore more 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 a memory stick is connected to a USB host, it will appear as a USB mass storage device, and the host will use a standard MSD class implementation to access the device.

### 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 supported by the device. 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.



## 2.4 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 burstly 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.

## 2.5 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 device. This part of the setup is called enumeration.

## 2.6 Product / Vendor IDs

Each USB device can be identified by its a Vendor and Product ID. A USB host does not have a Vendor and Product ID.

## 2.7 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.

# Chapter 3

# Running emUSB-Host on target hardware

This chapter explains how to integrate and run emUSB-Host on your target hardware.

# 3.1 Integrating emUSB-Host

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 Embedded Studio 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:

- Take a running project for your target hardware.
- Add emUSB-Host files to the project.
- Add hardware dependent configuration to the project.
- Prepare and run the application.

# 3.2 Take a running project

The project to start with should include the setup for basic hardware (e.g. CPU, PLL, DDR SDRAM) and initialization of the RTOS. 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.

# 3.3 Add emUSB-Host files

Add all necessary source files from the `USBH` folder to your project. You may simply add all files and let the linker drop everything not needed for your configuration. But there are some source files containing dependencies to emFile or embOS/IP. If you don't have these middleware components, remove the respective files from your project.

### Add RTOS layer

Additionally add the RTOS interface layer to your project. Choose a file from the folder `Sample/USBH/OS` that matches your RTOS. For embOS use `USBH_OS_embOS.c`.

### 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`
- `SEGGER`
- `USBH`

# 3.4 Configuring debugging output

While developing and testing emUSB-Host, we recommend to use the DEBUG configuration of emUSB-Host. This is enabled by setting the preprocessor symbol `DEBUG` to 1 (or `USBH_DEBUG` to 2). The DEBUG configuration contains many additional run-time checks and generate debug output messages which are very useful to identify problems that may occur during development. In case of a fatal problem (e.g. an invalid configuration) the program will end up in the function *`USBH_Panic()`* with a appropriate error message that describes the cause of the problem.

Add the file `USBH_ConfigIO.c` found in the folder `Config` to your project and configure it to match the message output method used by your debugging tools. If possible use RTT.

To later compile a release configuration, which has a significant smaller code footprint, simply set the preprocessor symbol `DEBUG` (or `USBH_DEBUG`) to 0.

# 3.5 Add hardware dependent configuration

To perform target hardware dependent runtime configuration, the emUSB-Host stack calls a function named `USBH_X_Config`. Typical tasks that may be done inside this function are:

- Assign memory to be used by the emUSB-Host stack.
- Select an appropriate driver for the USB host controller.
- Configure I/O pins of the MCU for USB.
- Configure PLL and clock divider necessary for USB operation.
- Install an interrupt service routine for USB.

Details can be found in *Runtime configuration* on page 372.

Sample configurations for popular evaluation boards are supplied with the driver shipment. They can be found in files called `USBH_Config_<TargetName>.c` in the folders `BSP/<BoardName>/Setup`.

Add the appropriate configuration file to your project. If there is no configuration file for your target hardware, take a file for a similar hardware and modify it if necessary.

If the file needs modifications, we recommend to copy it into the directory `Config` for easy updates to later versions of emUSB-Host.

### Add BSP file

Some targets require CPU specific functions for initialization, mainly for installing an interrupt service routine. They are contained in the file `BSP_USB.c`. Sample `BSP_USB.c` files for popular evaluation boards are supplied with the driver shipment. They can be found in the folders `BSP/<BoardName>/Setup`.

Add the appropriate `BSP_USB.c` file to your project. If there is no BSP file for your target hardware, take a file for a similar hardware and modify it if necessary.

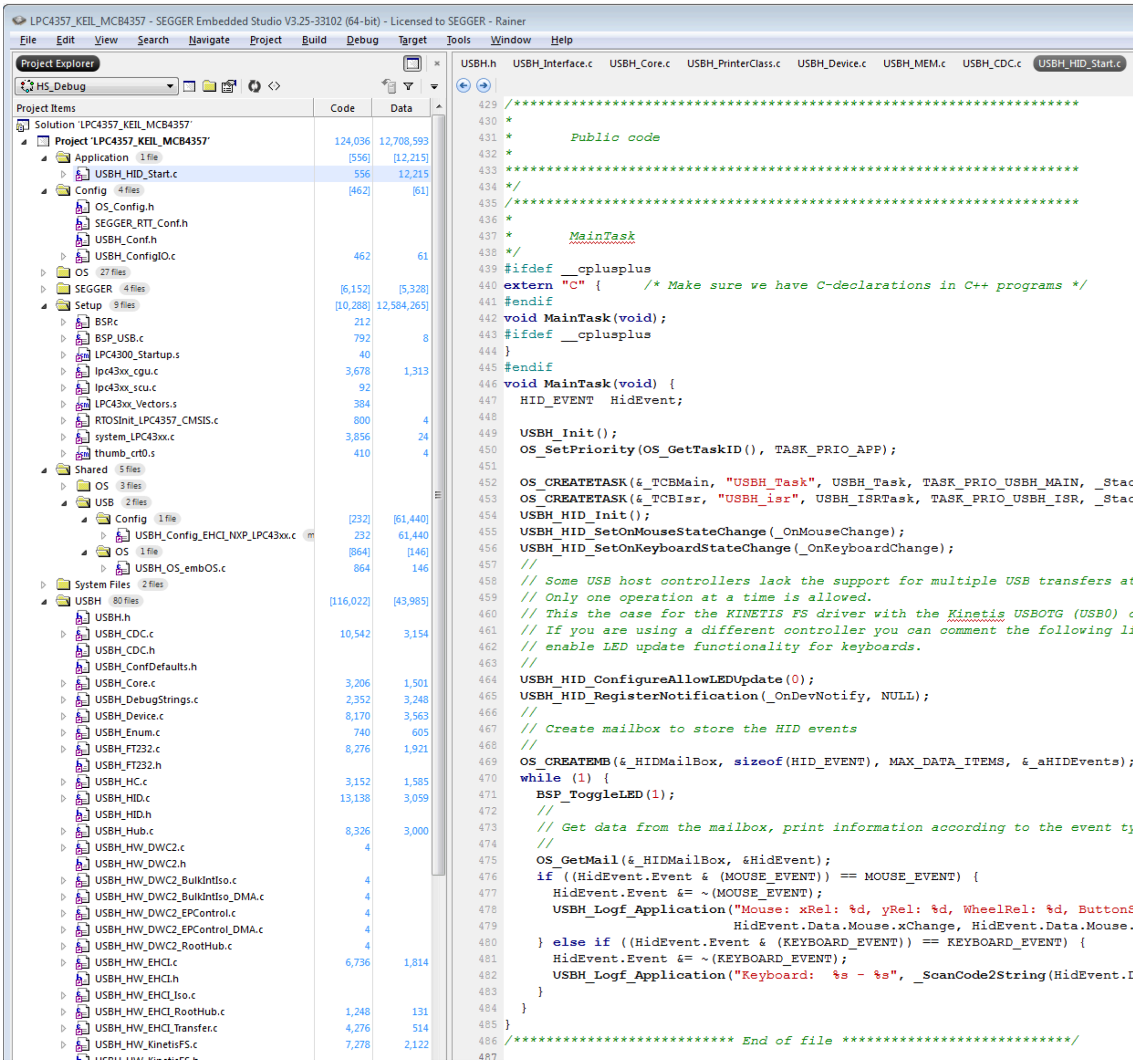
If the file needs modifications, we recommend to copy it into the directory `Config` for easy updates to later versions of emUSB-Host.

Note that a `BSP_USB.c` file is not always required, because for some target hardware all runtime configuration is done in `USBH_X_Config`.

# 3.6 Prepare and run the application

Choose a sample application from the folder `Application` and add it to your project. Sample applications are described in *Example applications* on page 33. Compile and run the application on the target hardware.




## Write your own application

Take one of the sample applications as a starting point to write your own application. In order to use emUSB-Host, the application has to:

- Initialize the USB core stack by calling `USBH_Init()`.
- Create two separate tasks that call the functions `USBH_Task()` and `USBH_ISRTask()`, respectively. Task priority requirements described in *Tasks and interrupt usage* on page 21 must be considered.
- Initialize the USB class drivers needed by calling the `USBH_<class>_Init()` function(s).

## 3.7 Updating emUSB-Host

If an existing project should be updated to a later emUSB-Host version, only files have to be replaced. You should have received the emUSB-Host update as a zip file. Unzip this file to the location of your choice and replace all emUSB-Host files in your project with the newer files from the emUSB-Host update shipment.

In general, all files from the following directories have to be updated:

- `USBH`
- `Inc`
- `SEGGER`
- `Doc`
- `Sample/USBH/OS`

Some files may contain modification required for project specific customization. These files should reside in the folder `Config` and must not be overwritten. This includes:

- `USBH_Conf.h`
- `USBH_ConfigIO.c`
- `BSP_USB.c`
- `USBH_Config_<TargetName>.c`

# Chapter 4

# Example applications

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

## 4.1 Overview

| File | Description |
|---|---|
| `USBH_HID_Start.c` | Demonstrates the handling of mouse and keyboard events. |
| `USBH_MSD_Start.c` | Demonstrates how to handle mass storage devices. |
| `USBH_Printer_Start.c` | Shows how to interact with a printer. |
| `USBH_CDC_Start.c` | Demonstrates communication with CDC devices. |
| `USBH_MTP_Start.c` | Shows how to interact with smart phones and other MTP-enabled devices. |
| `USBH_FT232_Start.c` | Demonstrates communication with FTDI serial adapters. |

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 (USBH_HID_Start.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 (USBH_MSD_Start.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. Information similar to the following is shown when a memory stick is connected:

```
<...>
**** Device added
Running sample on "msd:0:"

Reading volume information...

**** Volume information for msd:0:
125105536 KBytes total disk space
125105536 KBytes available free space

32768 bytes per cluster
3909548 clusters available on volume
3909548 free cluster available on volume

Creating file msd:0:\TestFile.txt...Ok
Contents of msd:0:
TESTFILE.TXT Attributes: A--- Size: 21

**** Unmount ****

Test with the following device was successful:
VendorId: 0x1234
ProductId: 0x5678
VendorName: XXXXXXX
ProductName: XXXXXXXXXXXXXX
Revision: 1.00
NumSectors: 250272768
BytesPerSector: 512
TotalSize: 122203 MByte
HighspeedCapable: No
ConnectedToRootHub: Yes
SelfPowered: No
Reported Imax: 500 mA
Connected to Port: 1
PortSpeed: FullSpeed
```

# 4.4 Printer interaction (USBH_Printer_Start.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 the 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
```

# 4.5 Serial communication (USBH_CDC_Start.c)

This example shows how to communicate with a CDC-enabled device. Since CDC is just a transport protocol it is not possible to write a generic sample which will work with all devices. This sample is designed to be used with a emUSB-Device CDC counterpart, the "USB_CDC_Echo.c" sample. It can also be used with any other device, but it may not be able to demonstrate continuous communication. The sample works as follows:

- When a CDC is connected the sample prints generic information about the device.
- After that the sample writes data onto the device.
- The sample reads data from the device and in case it has received any - sends it back.

With the emUSB-Device "USB_CDC_Echo.c" sample this causes a simple, continuous ping-pong of messages.

### Terminal output:

```
**** Device added
 <...>
 0:663 USBH_isr - INIT: USBH_ISRTask started

 **** Device added
 Vendor  Id = 0x1234
 Product Id = 0x5678
 Serial no. = 123456789
 Speed      = HighSpeed
 Started communication...
 <...>
 **** Device removed
```

# 4.6 Media Transfer Protocol (USBH_MTP_Start.c)

This example shows how to communicate with a MTP-enabled device. The sample demonstrates most of the emUSB-Host MTP API. When a MTP device is connected the sample prints generic information about the device. If the device is locked (e.g. pin code on a smart phone) the sample will wait for the user to unlock it. The sample will then iterate over the storages made available by the device, print information about it, print the file and folder list in the root directory and create a new file under it called "`SEGGER_Test.txt`".

**Terminal output:**

```
<...>
**** Device added
Vendor  Id            = 0x1234
Product Id            = 0x1234
Serial no.            = 1
Speed                 = FullSpeed
Manufacturer          : XXXXXX
Model                 : XXXXXXXXXXXXXXXXXXXXXX
DeviceVersion         : 8.10.12397.0
MTP SerialNumber      : 844848fb44583cbaa1ecae45545b3

USBH_MTP_CheckLock returns USBH_STATUS_ERROR
Please unlock the device to proceed.
<...>
_cbOnUserEvent: MTP Event received! EventCode: 0x4004, Para1: 0x00010001
                , Para2: 0x00000000, Para3: 0x00000000.
<...>
USBH_MTP_CheckLock returns USBH_STATUS_SUCCESS
Found storage with ID: 0
StorageType           = 0x0003
FilesystemType        = 0x0002
AccessCapability      = 0x0000
MaxCapacity           = 3959422976 bytes
FreeSpaceInBytes      = 1033814016 bytes
FreeSpaceInImages     = 0x00000000
StorageDescription    : Phone
VolumeLabel           : MTP Volume - 65537

Found 9 objects in directory 0xFFFFFFFF

Processing object 0x00000001 in directory 0xFFFFFFFF...
StorageID             = 0x00010001
ObjectFormat          = 0x3001
ParentObject          = 0xFFFFFFFF
ProtectionStatus      = 0x0001
Filename              : Documents
CaptureDate           : 20140522T0
ModificationDate      : 20160707T1

Processing object 0x00000002 in directory 0xFFFFFFFF...
StorageID             = 0x00010001
ObjectFormat          = 0x3001
ParentObject          = 0xFFFFFFFF
ProtectionStatus      = 0x0001
Filename              : Downloads
CaptureDate           : 20140522T0
ModificationDate      : 20160707T1
<...>
Creating new object with 135 bytes in folder 0xFFFFFFFF
with name SEGGER_Test.txt.
Created new object in folder 0xFFFFFFFF, ID: 0x000013F9.

Connection to MTP device closed.
<...>
**** Device removed
```

## 4.7 FTDI devices (USBH_FT232_Start.c)

This example shows how to communicate with a FTDI FT232 adapters. When a FT232 is connected the sample prints generic information about the device. After that it receives data from the connected FT232 adapter and sends it back. The sample is easily tested by using two identical FT232 adapters connected to each other via a null modem cable. One of the devices should be connected to emUSB-Host. The other to a PC. You can use any PC terminal emulator to send data from one adapter to the other, which will be then received by emUSB-Host and sent back. Baudrate and other serial setting should match between the sample and the PC for this to work.

### Terminal output:

```
**** Device added
 <...>
 3:213 MainTask - Vendor  Id = 0x0403
 3:213 MainTask - Product Id = 0x6001
 3:213 MainTask - bcdDevice  = 0x0600
 <...>
 **** 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 Target API

This section describes the functions that can be used by the target application.

| Function | Description |
|---|---|
| General | |
| USBH_Init() | Initializes the emUSB-Host stack. |
| USBH_Exit() | Shuts down and de-initializes the emUSB-Host stack. |
| USBH_ISRTask() | Processes the events triggered from the interrupt handler. |
| USBH_Task() | Manages the internal software timers. |
| USBH_IsRunning() | Returns whether the stack is running or not. |
| USBH_GetFrameNumber() | Retrieves the current frame number. |
| USBH_GetStatusStr() | Converts the result status into a string. |
| USBH_MEM_GetMaxUsed() | Returns the maximum used memory since initialization of the memory pool. |
| USBH_SetRootPortPower() | Set port of the root hub to a given power state. |
| USBH_HUB_SuspendResume() | Prepares hubs for suspend (stops the interrupt endpoint) or re-starts the interrupt endpoint functionality after a resume. |
| Runtime configuration | |
| USBH_AssignMemory() | Assigns a memory area that will be used by the memory management functions for allocating memory. |
| USBH_AssignTransferMemory() | Assigns a memory area for a heap that will be used for allocating DMA memory. |
| USBH_Config_SetV2PHandler() | Sets a virtual address to physical address translator. |
| USBH_ConfigPowerOnGoodTime() | Configures the default power on time that the host waits after connecting a device before starting to communicate with the device. |
| USBH_ConfigSupportExternalHubs() | Enable support for external USB hubs. |
| USBH_ConfigTransferBufferSize() | Configures the size of a copy buffer that can be used if the USB controller has limited access to the system memory. |
| USBH_SetCacheConfig() | Configures cache related functionality that might be required by the stack for several purposes such as cache handling in drivers. |
| USBH_SetOnSetPortPower() | Sets a callback for the set-port-power driver function. |
| Information about interfaces | |
| USBH_CreateInterfaceList() | Generates a list of available interfaces matching a given criteria. |
| USBH_DestroyInterfaceList() | Destroy a device list created by USBH_CreateInterfaceList() and free the related resources. |

| Function | Description |
|---|---|
| USBH_GetInterfaceId() | Returns the interface id for a specified interface. |
| USBH_GetInterfaceInfo() | Obtain information about a specified interface. |
| USBH_GetInterfaceSerial() | Retrieves the serial number of the device containing the given interface. |
| USB interface handling | |
| USBH_CloseInterface() | Close an interface handle that was opened with USBH_OpenInterface(). |
| USBH_GetCurrentConfigurationDescriptor() | Retrieves the current configuration descriptor of the device containing the given interface. |
| USBH_GetDeviceDescriptor() | Retrieves the current device descriptor of the device containing the given interface. |
| USBH_GetEndpointDescriptor() | Retrieves an endpoint descriptor of the device containing the given interface. |
| USBH_GetInterfaceDescriptor() | Retrieves the interface descriptor of the given interface. |
| USBH_GetInterfaceIdByHandle() | Get the interface ID for a given index. |
| USBH_GetSerialNumber() | Retrieves the serial number of the device containing the given interface. |
| USBH_GetSpeed() | Returns the operating speed of the device. |
| USBH_OpenInterface() | Opens the specified interface. |
| USBH_GetPortInfo() | Obtains information about a connected USB device. |
| USBH_SubmitUrb() | Submits an URB. |
| Notification | |
| USBH_RegisterEnumErrorNotification() | Registers a notification for a port enumeration error. |
| USBH_RegisterPnPNotification() | Registers a notification function for PnP events. |
| USBH_RestartEnumError() | Restarts the enumeration process for all devices that have failed to enumerate. |
| USBH_UnregisterEnumErrorNotification() | Removes a registered notification for a port enumeration error. |
| USBH_UnregisterPnPNotification() | Removes a previously registered notification for PnP events. |

## 5.1.1 USBH_AssignMemory()

### Description

Assigns a memory area that will be used by the memory management functions for allocating memory. This function must be called in the initialization phase.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pMem | Pointer to the memory area. |
| NumBytes | Size of the memory area in bytes. |

### Additional information

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

For some USB host controllers additionally a separate memory heap for DMA memory must be provided by calling `USBH_AssignTransferMemory()`.

## 5.1.2 USBH_AssignTransferMemory()

### Description

Assigns a memory area for a heap that will be used for allocating DMA memory. This function must be called in the initialization phase.

The memory area provided to this function must fulfill the following requirements:

- Not cachable/bufferable.
- Fast access to avoid timeouts.
- USB-Host controller must have full read/write access.
- Cache aligned

If the physical address is not equal to the virtual address of the memory area (address translation by an MMU), additionally a mapping function must be installed using `USBH_Config_SetV2PHandler()`.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pMem | Pointer to the memory area (virtual address). |
| NumBytes | Size of the memory area in bytes. |

### 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.

## 5.1.3 USBH_CloseInterface()

### Description

Close an interface handle that was opened with `USBH_OpenInterface()`.

### Prototype

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

### Parameters

| Parameter | Description |
| --- | --- |
| hInterface | Handle to a valid interface, returned by `USBH_OpenInterface()`. |

### Additional information

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

## 5.1.4 USBH_Config_SetV2PHandler()

### Description

Sets a virtual address to physical address translator. Is required, if the physical address is not equal to the virtual address of the memory used for DMA access (address translation by an MMU). See `USBH_AssignTransferMemory`.

### Prototype

```
void USBH_Config_SetV2PHandler(USBH_V2P_FUNC * pfV2PHandler);
```

### Parameters

| Parameter | Description |
| --- | --- |
| `pfV2PHandler` | Handler to be called to convert virtual address. |

## 5.1.5 USBH_ConfigPowerOnGoodTime()

### Description

Configures the default power on time that the host waits after connecting a device before starting to communicate with the device. The default value is 300 ms.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| PowerGoodTime | Time the stack should wait before doing any other operation (im ms). |

### Additional information

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

## 5.1.6 USBH_ConfigSupportExternalHubs()

### Description

Enable support for external USB hubs.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| OnOff | 1 - Enable support for external hubs 0 - Disable support for external hubs |

### Additional information

This function should not be called if no external hub support is required to avoid the code for external hubs to be linked into the application.

## 5.1.7 USBH_ConfigTransferBufferSize()

### Description

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

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller. |
| Size | Size of the buffer in bytes. Must be a multiple of the maximum packet size (512 for high speed, 64 for full speed). |

## 5.1.8 USBH_CreateInterfaceList()

### Description

Generates a list of available interfaces matching a given criteria.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pInterfaceMask | Pointer to a caller provided structure, that allows to select interfaces to be included in the list. If this pointer is NULL all available interfaces are returned. |
| pInterfaceCount | Pointer to a variable that receives the number of interfaces in the list created. |

### 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);
        _ShowIfaceInfo(IfaceId);
      }
    }
    //
    // Ensure the list is properly cleaned up
    //
    USBH_DestroyInterfaceList(hIfaceList);
  }
}
```

## 5.1.9 USBH_DestroyInterfaceList()

### Description

Destroy a device list created by `USBH_CreateInterfaceList()` and free the related resources.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterfaceList | Valid handle to a interface list, returned by `USBH_CreateInterfaceList()`. |

## 5.1.10 USBH_Exit()

### Description

Shuts down and de-initializes the emUSB-Host stack. All resources will be freed within this function. This includes also the removing and deleting of all host controllers.

Before this function can be used, the exit functions of all initialized USB classes (e.g. `USBH_CDC_Exit()`, `USBH_MSD_Exit()`, ...) must be called.

Calling `USBH_Exit()` will cause the functions `USBH_Task()` and `USBH_ISRTask()` to return.

### Prototype

```
void USBH_Exit(void);
```

### Additional information

After this function call, no other function of the USB stack should be called.

## 5.1.11 USBH_GetCurrentConfigurationDescriptor()

### Description

Retrieves the current configuration descriptor of the device containing the given interface.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| pDescriptor | Pointer to a buffer where the descriptor is stored. |
| pBufferSize | • in Size of the buffer pointed to by pDescriptor.<br>• out Number of bytes copied into the buffer. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

The function returns a copy of the current configuration descriptor, that was stored during the device enumeration. If the given buffer size is too small the configuration descriptor returned is truncated.

## 5.1.12 USBH_GetDeviceDescriptor()

### Description

Retrieves the current device descriptor of the device containing the given interface.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| pDescriptor | Pointer to a buffer where the descriptor is stored. |
| pBufferSize | • in Size of the buffer pointed to by pDescriptor.<br>• out Number of bytes copied into the buffer. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

The function returns a copy of the current device descriptor, that was stored during the device enumeration. If the given buffer size is too small the device descriptor returned is truncated.

## 5.1.13 USBH_GetEndpointDescriptor()

### Description

Retrieves an endpoint descriptor of the device containing the given interface.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by `USBH_OpenInterface()`. |
| 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`, that specifies the endpoint selection pattern. |
| pBuffer | Pointer to a buffer where the descriptor is stored. |
| pBufferSize | • in Size of the buffer pointed to by pBuffer.<br>• out Number of bytes copied into the buffer. |

### Return value

`USBH_STATUS_SUCCESS` on success. Other values indicate an error.

### Additional information

The endpoint descriptor is extracted from the current configuration descriptor, that was stored during the device enumeration. If the given buffer size is too small the endpoint descriptor returned is truncated.

# 5.1.14 USBH_GetFrameNumber()

### Description

Retrieves the current frame number.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| pFrameNumber | Pointer to a variable that receives the frame number. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### 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.

CAUTION: The functionality is not implemented for all host drivers. For some host controllers the function may always return a frame number of 0.

## 5.1.15 USBH_GetInterfaceDescriptor()

### Description

Retrieves the interface descriptor of the given interface.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| AlternateSetting | Specifies the alternate setting for this interface. |
| pBuffer | Pointer to a buffer where the descriptor is stored. |
| pBufferSize | • in Size of the buffer pointed to by pBuffer.<br>• out Number of bytes copied into the buffer. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

The interface descriptor is extracted from the current configuration descriptor, that was stored during the device enumeration. The interface descriptor belongs to the interface that is identified by hInterface. If the interface has different alternate settings the interface descriptors of each alternate setting can be requested.

If the given buffer size is too small the interface descriptor returned is truncated.

## 5.1.16 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 | Valid handle to a interface list, returned by USBH_CreateInterfaceList(). |
| Index | Specifies the zero based index for an interface in the 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.

### Example

See *USBH_CreateInterfaceList* on page 51.

## 5.1.17 USBH_GetInterfaceIdByHandle()

### Description

Get the interface ID for a given index. A returned value of zero indicates an error.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Handle to a valid interface, returned by USBH_OpenInterface(). |
| pInterfaceId | Pointer to a variable that will receive the interface id. |

### Return value

USBH_STATUS_SUCCESS on success. Any other value means error.

### 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.

## 5.1.18 USBH_GetInterfaceInfo()

### Description

Obtain 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 that will receive the interface information on success. |

### Return value

USBH_STATUS_SUCCESS on success. Any other value means error.

### 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.

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.

## 5.1.19 USBH_GetInterfaceSerial()

### Description

Retrieves the serial number of the device containing the given interface.

### Prototype

```
USBH_STATUS USBH_GetInterfaceSerial(USBH_INTERFACE_ID   InterfaceID,
                                    U32                 BuffSize,
                                    U8                * pSerialNumber,
                                    U32               * pSerialNumberSize);
```

### Parameters

| Parameter | Description |
|---|---|
| InterfaceID | Id of the interface to query. |
| BuffSize | Size of the buffer pointed to by pSerialNumber. |
| pSerialNumber | Pointer to a buffer where the serial number is stored. |
| pSerialNumberSize | out Number of bytes copied into the buffer. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

The serial number is returned as a UNICODE string in USB little endian format. The number of valid bytes is returned in pSerialNumberSize. The string is not zero terminated. The returned data does not contain a USB descriptor header and is encoded in the first language Id. This string is a copy of the serial number string that was requested during the enumeration. If the device does not support a USB serial number string the function returns USBH_STATUS_SUCCESS and a length of 0. If the given buffer size is too small the serial number returned is truncated.

## 5.1.20 USBH_GetPortInfo()

### Description

Obtains information about a connected USB device.

### Prototype

```
USBH_STATUS USBH_GetPortInfo(USBH_INTERFACE_ID   InterfaceID,
                             USBH_PORT_INFO    * pPortInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| InterfaceID | Id of an interface of the device to query. |
| pPortInfo | Pointer to a caller allocated structure that will receive the port information on success. |

### Return value

USBH_STATUS_SUCCESS on success. Any other value means error.

## 5.1.21 USBH_GetSerialNumber()

### Description

Retrieves the serial number of the device containing the given interface.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| pBuffer | Pointer to a buffer where the serial number is stored. |
| pBufferSize | • in Size of the buffer pointed to by pBuffer.<br>• out Number of bytes copied into the buffer. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

The serial number is returned as a UNICODE string in USB little endian format. The number of valid bytes is returned in pBufferSize. The string is not zero terminated. The returned data does not contain a USB descriptor header and is encoded in the first language Id. This string is a copy of the serial number string that was requested during the enumeration. If the device does not support a USB serial number string the function returns USBH_STATUS_SUCCESS and a length of 0. If the given buffer size is too small the serial number returned is truncated.

## 5.1.22 USBH_GetSpeed()

### Description

Returns the operating speed of the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Valid handle to an interface, returned by USBH_OpenInterface(). |
| pSpeed | Pointer to a variable that will receive the speed information. |

### Return value

USBH_STATUS_SUCCESS on success. Other values indicate an error.

### Additional information

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

## 5.1.23 USBH_GetStatusStr()

### Description

Converts the result status into a string.

### Prototype

```
char *USBH_GetStatusStr(USBH_STATUS x);
```

### Parameters

| Parameter | Description |
|---|---|
| x | Result status to convert. |

### Return value

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

## 5.1.24 USBH_ISRTask()

### Description

Processes the events triggered from the interrupt handler. This function must run as a separate task in order to use the emUSBH stack. The functions only returns, if the USBH stack is shut down (if `USBH_Exit()` was called). In order for the emUSB-Host to work reliably, the task should have the highest priority of all tasks dealing with USB.

### Prototype

```
void USBH_ISRTask(void);
```

### Additional information

This function waits for events from the interrupt handler of the host controller and processes them.

When `USBH_Exit()` is used in the application this function should not be directly started as a task, as it returns when `USBH_Exit()` is called. A wrapper task can be used in this case, see `USBH_IsRunning()` for a sample.

### Note

Task priority requirements described in *Tasks and interrupt usage* on page 21 must be considered.

## 5.1.25 USBH_Init()

### Description

Initializes the emUSB-Host stack.

### Prototype

```
void USBH_Init(void);
```

### Additional information

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

## 5.1.26 USBH_MEM_GetMaxUsed()

### Description

Returns the maximum used memory since initialization of the memory pool.

### Prototype

```
U32 USBH_MEM_GetMaxUsed(int Idx);
```

### Parameters

| Parameter | Description |
| --- | --- |
| Idx | Index of memory pool.<br>• 0 - normal memory<br>• 1 - transfer memory. |

### Return value

Maximum used memory in bytes.

### Additional information

This function only works in a debug configuration of emUSB-Host. If compiled as release configuration, this function always returns 0.

## 5.1.27 USBH_SetRootPortPower()

### Description

Set port of the root hub to a given power state.

The application must ensure that no transaction is pending on the port before setting it into suspend state.

### Prototype

```
void USBH_SetRootPortPower(U32              HCIndex,
                           U8               Port,
                           USBH_POWER_STATE State);
```

### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller. |
| Port | Port number of the roothub. Ports are counted starting with 1. if set to 0, the new state is set to all ports of the root hub. |
| State | New power state of the port. |

# 5.1.28 USBH_HUB_SuspendResume()

### Description

Prepares hubs for suspend (stops the interrupt endpoint) or re-starts the interrupt endpoint functionality after a resume. All hubs connected to a given port of a host controller (directly or indirectly) are handled by the function.

### Prototype

```
void USBH_HUB_SuspendResume(U32 HCIndex,
                            U8  Port,
                            U8  State);
```

### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller. |
| Port | Port number of the roothub. Ports are counted starting with 1. if set to 0, the function applies to all ports of the root hub. |
| State | 0 - Prepare for suspend. 1 - Return from resume. |

### Additional information

The application must make sure no transactions are running when setting a device into suspend mode. This function is used in combination with USBH_SetRootPortPower(). Call this function before USBH_SetRootPortPower(x, y, USBH_SUSPEND) with State = 0. Call this function after USBH_SetRootPortPower(x, y, USBH_NORMAL_POWER) with State = 1;

## 5.1.29 USBH_OpenInterface()

### Description

Opens the specified interface.

### Prototype

```
USBH_STATUS USBH_OpenInterface(USBH_INTERFACE_ID       InterfaceID,
                               U8                       Exclusive,
                               USBH_INTERFACE_HANDLE * pInterfaceHandle);
```

### 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. |
| pInterfaceHandle | Pointer where the handle to the opened interface is stored. |

### Return value

USBH_STATUS_SUCCESS on success. Any other value means error.

### Additional information

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

If the interface is allocated exclusive no other application can open it.

## 5.1.30 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 * pfEnumErrorCallback);
```

### Parameters

| Parameter | Description |
|---|---|
| pContext | 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 that is called if a port enumeration error occurs. |

### 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_UnregisterEnumErrorNotification() 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. The callback routine must not block.

## 5.1.31 USBH_RegisterPnPNotification()

### Description

Registers a notification function for PnP events.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pPnPNotification | Pointer to a caller provided structure. |

### Return value

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

### Additional information

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.

## 5.1.32 USBH_RestartEnumError()

### Description

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

### Prototype

```
void USBH_RestartEnumError(void);
```

### Additional information

The USB stack retries each enumeration again until the default retry count is reached.

## 5.1.33 USBH_SetCacheConfig()

### Description

Configures cache related functionality that might be required by the stack for several purposes such as cache handling in drivers.

### Prototype

```
void USBH_SetCacheConfig(const SEGGER_CACHE_CONFIG * pConfig,
                               unsigned              ConfSize);
```

### Parameters

| Parameter | Description |
|---|---|
| pConfig | Pointer to an element of SEGGER_CACHE_CONFIG . |
| ConfSize | Size of the passed structure in case library and header size of the structure differs. |

### Additional information

This function has to called in USBH_X_Config().

## 5.1.34 USBH_SetOnSetPortPower()

### Description

Sets a callback for the set-port-power driver function. The user callback is called when the ports are added to the host driver instance, this occurs during initialization, or when the ports are removed (during de-initialization). Using this function is necessary if the port power is not controlled directly through the USB controller but is provided from an external source.

### Prototype

```
void USBH_SetOnSetPortPower(USBH_ON_SETPORTPOWER_FUNC * pfOnSetPortPower);
```

### Parameters

| Parameter | Description |
|---|---|
| pfOnSetConfig | Pointer to a user-provided callback function of type USBH_ON_SETPORTPOWER_FUNC. |

### Additional information

The callback function should not block.

# 5.1.35 USBH_SubmitUrb()

### Description

Submits an URB. Interface function for all asynchronous requests.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hInterface | Handle to a interface. |
| pUrb | Pointer to a caller allocated structure.<br>• in The URB which should be submitted.<br>• out 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. |

### Return value

The request can fail for different reasons. In that case the return value is different 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 driver. The storage of the URB must not be freed nor modified as long as the ownership is assigned to the driver. The 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.

### Notes

A pending URB transactions may be aborted with an abort request by using USBH_SubmitUrb with a new URB where Urb->Header.Function = USBH_FUNCTION_ABORT_ENDPOINT and Urb->Request.EndpointRequest.Endpoint = EndpointAddressToAbort. Otherwise this operation will last until the device has responded to the request or the device has been disconnected.

### Example (asynchronous operation)

```
static U8       _Buffer[512];
static USBH_URB _Urb;


static void _OnUrbCompletion(USBH_URB * pUrb) {
  if (pUrb->Header.Status == USBH_SUCCESS) {
    ProcessData(pUrb->BulkIntRequest.pBuffer, pUrb->BulkIntRequest.Lenght);
  } else {
    // error handling ...
  }
}
```

```
//
// Start IN transfer on interface 'hInterface' for endpoint 'Ep'
//
_Urb.Header.Function          = USBH_FUNCTION_BULK_REQUEST;
_Urb.Header.pfOnCompletion    = _OnUrbCompletion;
_Urb.Header.pContext          = NULL;
_Urb.BulkIntRequest.pBuffer   = &_Buffer[0];
_Urb.BulkIntRequest.Lenght    = sizeof(_Buffer);
_Urb.BulkIntRequest.Endpoint = Ep;
Status = USBH_SubmitUrb(hInterface, pUrb);
if (Status != USBH_STATUS_PENDING) {
  // error handling ...
}
```

### Example (synchronous operation)

```
static U8                 _Buffer[512];
static USBH_URB           _Urb;

static void _OnUrbCompletion(USBH_URB * pUrb) {
  USBH_OS_EVENT_OBJ *pEvent;

  pEvent = (USBH_OS_EVENT_OBJ *)pUrb->Header.pContext;
  USBH_OS_SetEvent(pEvent);
}


USBH_OS_EVENT_OBJ *pEvent;
//
// Start IN transfer on interface 'hInterface' for endpoint 'Ep'
//
pEvent = USBH_OS_AllocEvent();
_Urb.Header.Function          = USBH_FUNCTION_BULK_REQUEST;
_Urb.Header.pfOnCompletion    = _OnUrbCompletion;
_Urb.Header.pContext          = pEvent;
_Urb.BulkIntRequest.pBuffer   = &_Buffer[0];
_Urb.BulkIntRequest.Lenght    = sizeof(_Buffer);
_Urb.BulkIntRequest.Endpoint = Ep;
Status = USBH_SubmitUrb(hInterface, pUrb);
if (Status != USBH_STATUS_PENDING) {
  // error handling ...
} else {
  USBH_OS_WaitEvent(pEvent);
  if (_Urb.Header.Status == USBH_SUCCESS) {
    ProcessData(_Urb.BulkIntRequest.pBuffer, _Urb.BulkIntRequest.Lenght);
  } else {
    // error handling ...
  }
}
USBH_OS_FreeEvent(pEvent);
```

## 5.1.36 USBH_Task()

### Description

Manages the internal software timers. This function must run as a separate task in order to use the emUSBH stack. The functions only returns, if the USBH stack is shut down (if `USBH_Exit()` was called).

### Prototype

```
void USBH_Task(void);
```

### Additional information

The function iterates over the list of active timers and invokes the registered callback functions in case the timer expired.

When `USBH_Exit()` is used in the application this function should not be directly started as a task, as it returns when `USBH_Exit()` is called. A wrapper task can be used in this case, see `USBH_IsRunning()` for a sample.

### Note

Task priority requirements described in *Tasks and interrupt usage* on page 21 must be considered.

## 5.1.37 USBH_IsRunning()

### Description

Returns whether the stack is running or not.

### Prototype

```
int USBH_IsRunning(void);
```

### Return value

| | |
|---|---|
| 0 | USBH is not running |
| 1 | USBH is running |

### Example

```
/*********************************************************************
*
*       _USBH_Task
*
*  Function description
*    Wrapper task for emUSBH USBH_Task.
*    Before the function is called, the task stays in a loop to
*    check whether the emUSBH stack is running.
*/
static void _USBH_Task(void) {
  while (1) {
    //
    // Wait until USBH is Ready
    //
    while (USBH_IsRunning() == 0) {
      OS_Delay(10);
    }
    USBH_Task();
  }
}

/*********************************************************************
*
*       _USBH_ISRTask
*
*  Function description
*    Wrapper task for emUSBH USBH_ISRTask.
*    Before the function is called, the task stays in a loop to
*    check whether the emUSBH stack is running.
*/
static void _USBH_ISRTask(void) {
  while (1) {
    //
    // Wait until USBH is Ready
    //
    while (USBH_IsRunning() == 0) {
      OS_Delay(10);
    }
    USBH_ISRTask();
  }
}
```

## 5.1.38 USBH_UnregisterEnumErrorNotification()

### Description

Removes a registered notification for a port enumeration error.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hEnumError | A valid handle for the notification previously returned from USBH_RegisterEnumErrorNotification(). |

### Additional information

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

## 5.1.39 USBH_UnregisterPnPNotification()

### Description

Removes a previously registered notification for PnP events.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hNotification | A valid handle for a PnP notification previously registered by a call to USBH_RegisterPnPNotification(). |

### Additional information

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

## 5.2 Data structures

The table below lists the available data structures.

| Structure | Description |
| --- | --- |
| USBH_BULK_INT_REQUEST | Defines parameters for a BULK or INT transfer request. |
| USBH_CONTROL_REQUEST | Defines parameters for a CONTROL transfer request. |
| USBH_ENDPOINT_REQUEST | Defines parameter for an endpoint operation. |
| USBH_ENUM_ERROR | Is used as a notification parameter for the USBH_ON_ENUM_ERROR_FUNC callback function. |
| USBH_EP_MASK | Is used as an input parameter to get an endpoint descriptor. |
| USBH_HEADER | Common parameters for all URB based requests. |
| USBH_INTERFACE_INFO | This structure contains information about a USB interface and the related device and is returned by the function USBH_GetInterfaceInfo(). |
| USBH_INTERFACE_MASK | Data structure that defines conditions to select USB interfaces. |
| USBH_PNP_NOTIFICATION | Is used as an input parameter for the USBH_RegisterEnumErrorNotification() function. |
| USBH_PORT_INFO | Information about a connected USB device returned by USBH_GetPortInfo(). |
| USBH_SET_INTERFACE | Defines parameters for a control request to set an alternate interface setting. |
| USBH_SET_POWER_STATE | Defines parameters to set or reset suspend mode for a device. |
| USBH_URB | This data structure is used to submit an URB. |
| SEGGER_CACHE_CONFIG | Used to pass cache configuration and callback function pointers to the stack. |

# 5.2.1 USBH_BULK_INT_REQUEST

## Description

Defines parameters for a BULK or INT transfer request. Used with USBH_FUNCTION_BULK_REQUEST and USBH_FUNCTION_INT_REQUEST.

## Type definition

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

## Structure members

| Member | Description |
|---|---|
| Endpoint | Specifies the endpoint address with direction bit. |
| pBuffer | Pointer to a caller provided buffer. |
| Length | • in length of data / size of buffer (in bytes).<br>• out Bytes transferred. |

## 5.2.2 USBH_CONTROL_REQUEST

### Description

Defines parameters for a CONTROL transfer request. Used with USBH_FUNCTION_CONTROL_REQUEST.

### Type definition

```
typedef struct {
  USBH_SETUP_PACKET  Setup;
  U8                 Endpoint;
  void             * pBuffer;
  U32                Length;
} USBH_CONTROL_REQUEST;
```

### Structure members

| Member | Description |
|---|---|
| Setup | The setup packet, direction of data phase, the length field must be valid! |
| Endpoint | The endpoint address with direction bit. Use 0 for default endpoint. |
| pBuffer | Pointer to the caller provided storage, 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. |

### Additional information

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 any setup packet can be submitted. Some standard requests, like SetAddress can be sent but would lead to a breakdown of the communication. It is not allowed to set the following standard requests:

***SetAddress***: It is assigned by the USB stack 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***

## 5.2.3 USBH_ENDPOINT_REQUEST

### Description

Defines parameter for an endpoint operation. Used with USBH_FUNCTION_ABORT_ENDPOINT and USBH_FUNCTION_RESET_ENDPOINT.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| Endpoint | Specifies the endpoint address with direction bit. |

## 5.2.4 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.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| Flags | Additional flags to determine the location and the type of the error.<br>• USBH_ENUM_ERROR_EXTHUBPORT_FLAG means the device is connected to an external hub.<br>• USBH_ENUM_ERROR_RETRY_FLAG the bus driver retries the enumeration of this device automatically.<br>• 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.<br>• 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.<br>• USBH_ENUM_ERROR_ROOT_PORT_RESET means an error during the USB reset of a root hub port occurs.<br>• USBH_ENUM_ERROR_HUB_PORT_RESET means an error during a reset of an external hub port occurs. |
| 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. |

## 5.2.5 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.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| Mask | This member contains the information which fields are valid. It is an OR combination of the following flags:<br>• USBH_EP_MASK_INDEX The Index is used for comparison.<br>• USBH_EP_MASK_ADDRESS The Address field is used for comparison.<br>• USBH_EP_MASK_TYPE The Type field is used for comparison.<br>• 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 contains the type of the endpoint:<br>• USB_EP_TYPE_CONTROL<br>• USB_EP_TYPE_BULK<br>• USB_EP_TYPE_ISO<br>• USB_EP_TYPE_INT |
| Direction | If valid, this member specifies a direction. It is one of the following values:<br>• USB_IN_DIRECTION From device to host<br>• USB_OUT_DIRECTION From host to device |

# 5.2.6 USBH_HEADER

### Description

Common parameters for all URB based requests.

### Type definition

```
typedef struct {
  USBH_FUNCTION                 Function;
  USBH_STATUS                   Status;
  USBH_ON_COMPLETION_FUNC     * pfOnCompletion;
  void                        * pContext;
  void                        * pUbdContext;
  USBH_ON_COMPLETION_FUNC     * pfOnInternalCompletion;
  void                        * pInternalContext;
  U32                           HcFlags;
  USBH_ON_COMPLETION_USER_FUNC * pfOnUserCompletion;
  void                        * pUserContext;
  USB_DEVICE                  * pDevice;
} USBH_HEADER;
```

### Structure members

| Member | Description |
|---|---|
| Function | Function code defines the operation of the URB. |
| 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 | This member can be used by the caller to store a context passed to the completion routine. |
| pUbdContext | Internal use. |
| pfOnInternalCompletion | Internal use. |
| pInternalContext | Internal use. |
| HcFlags | Internal use. |
| pfOnUserCompletion | Internal use. |
| pUserContext | Internal use. |
| pDevice | Internal use. |

## 5.2.7 USBH_INTERFACE_INFO

### Description

This structure contains information about a USB interface and the related device and is returned by the function USBH_GetInterfaceInfo().

### Type definition

```
typedef struct {
  USBH_INTERFACE_ID  InterfaceId;
  USBH_DEVICE_ID     DeviceId;
  U16                VendorId;
  U16                ProductId;
  U16                bcdDevice;
  U8                 Interface;
  U8                 Class;
  U8                 SubClass;
  U8                 Protocol;
  unsigned int       OpenCount;
  U8                 ExclusiveUsed;
  USBH_SPEED         Speed;
  U8                 SerialNumberSize;
  U8                 NumConfigurations;
  U8                 CurrentConfiguration;
  U8                 HCIndex;
  U8                 AlternateSetting;
} USBH_INTERFACE_INFO;
```

### Structure members

| Member | Description |
|---|---|
| InterfaceId | 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 | 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 | The Vendor ID of the device. |
| ProductId | The Product ID of the device. |
| bcdDevice | The BCD coded device version. |
| Interface | The USB interface number. |
| Class | The interface class. |
| SubClass | The interface sub class. |
| Protocol | The interface protocol. |
| OpenCount | Number of open handles for this interface. |
| ExclusiveUsed | If not 0, this interface is used exclusively. |
| Speed | Operation speed of the device. |
| SerialNumberSize | The size of the serial number in bytes, 0 means not available or error during request. The serial number itself can be retrieved using USBH_GetInterfaceSerial(). |
| NumConfigurations | Number of different configuration of the device. |

| Member | Description |
|---|---|
| `CurrentConfiguration` | Currently selected configuration, zero-based: 0…(`NumConfigurations`-1) |
| `HCIndex` | Index of the host controller the device is connected to. |
| `AlternateSetting` | The current alternate setting for this interface. |

## 5.2.8 USBH_INTERFACE_MASK

### Description

Data structure that defines conditions to select USB interfaces. Can be used to register notifications. Members that are not seleted with Mask need not be initialized.

### Type definition

```
typedef struct {
  U16         Mask;
  U16         VendorId;
  U16         ProductId;
  U16         bcdDevice;
  U8          Interface;
  U8          Class;
  U8          SubClass;
  U8          Protocol;
  const U16 * pVendorIds;
  const U16 * pProductIds;
  U16         NumIds;
} USBH_INTERFACE_MASK;
```

### Structure members

| Member | Description |
|---|---|
| Mask | Contains an OR combination of the following flags. If the flag is set the related member of this structure is compared to the properties of the USB interface.<br>USBH_INFO_MASK_VID Compare the Vendor ID (VID) of the device.<br>USBH_INFO_MASK_PID Compare the Product ID (PID) of the device.<br>USBH_INFO_MASK_DEVICE Compare the bcdDevice value of the device.<br>USBH_INFO_MASK_INTERFACE Compare the interface number.<br>USBH_INFO_MASK_CLASS Compare the class of the interface.<br>USBH_INFO_MASK_SUBCLASS Compare the sub class of the interface.<br>USBH_INFO_MASK_PROTOCOL Compare the protocol of the interface.<br>USBH_INFO_MASK_VID_ARRAY Compare the Vendor ID (VID) of the device to a list if ids.<br>USBH_INFO_MASK_PID_ARRAY Compare the Product ID (PID) of the device to a list if ids.<br>If both USBH_INFO_MASK_VID_ARRAY and USBH_INFO_MASK_PID_ARRAY are selected, then the VendorId/ProductId of the device is compared to pairs pVendorIds[i]/pProductIds[i]. |
| 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. |
| pVendorIds | Points to an array of Vendor IDs. |

| Member | Description |
|---|---|
| pProductIds | Points to an array of Product IDs. |
| NumIds | Number of ids in *pVendorIds and *pProductIds. When only USBH_INFO_MASK_VID_ARRAY is set this is the size of the pVendorIds array. When only USBH_INFO_MASK_PID_ARRAY is set this is the size of the pProductIds array. When both are set this is the size for both arrays (the arrays have to be the same size when both flags are set). |

## 5.2.9 USBH_PNP_NOTIFICATION

### Description

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

### Type definition

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

### Structure members

| Member | Description |
| --- | --- |
| pfPnpNotification | The notification function that is called from the USB stack if a PnP event occurs. |
| pContext | Pointer to a context, that is passed to the notification function. |
| InterfaceMask | Mask for the interfaces for which the PnP notifiation should be called. |

## 5.2.10 USBH_PORT_INFO

### Description

Information about a connected USB device returned by `USBH_GetPortInfo()`.

### Type definition

```
typedef struct {
  U8                  IsHighSpeedCapable;
  U8                  IsRootHub;
  U8                  IsSelfPowered;
  U8                  HCIndex;
  U16                 MaxPower;
  U16                 PortNumber;
  USBH_SPEED          PortSpeed;
  USBH_DEVICE_ID      DeviceId;
  USBH_DEVICE_ID      HubDeviceId;
  USBH_INTERFACE_ID   HubInterfaceId;
} USBH_PORT_INFO;
```

### Structure members

| Member | Description |
|---|---|
| IsHighSpeedCapable | • 1: Port supports high-speed, full-speed and low-speed communication.<br>• 0: Port supports only full-speed and low-speed communication. |
| IsRootHub | • 1: RootHub, device is directly connected to the host.<br>• 0: Device is connected via an external hub to the host. |
| IsSelfPowered | • 1: Device is externally powered<br>• 0: Device is powered by USB host controller. |
| HCIndex | Index of the host controller the device is connected to. |
| MaxPower | Max power the USB device consumes from USB host controller / USB hub in mA. |
| PortNumber | Port number of the hub or roothub. Ports are counted starting with 1. |
| PortSpeed | The port speed is the speed with which the device is connected. Can be either USBH_LOW_SPEED or USBH_FULL_SPEED or USBH_HIGH_SPEED. |
| DeviceId | 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. |
| HubDeviceId | The unique device Id of the HUB, if the device is connected via an external HUB. If IsRootHub = 1, then HubDeviceId is zero. |
| HubInterfaceId | Interface Id of the HUB, if the device is connected via an external HUB. If IsRootHub = 1, then HubInterfaceId is zero. |

# 5.2.11 USBH_SET_INTERFACE

### Description

Defines parameters for a control request to set an alternate interface setting. Used with USBH_FUNCTION_SET_INTERFACE.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| AlternateSetting | Number of alternate interface setting (zero based). |

## 5.2.12 USBH_SET_POWER_STATE

### Description

Defines parameters to set or reset suspend mode for a device. Used with USBH_FUNCTION_SET_POWER_STATE.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| PowerState | New power state of the device. |

### Additional information

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

## 5.2.13 USBH_URB

### Description

This data structure is used to submit an URB. The URB is the basic structure for all asynchronous operations on the USB stack. 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.

### Prototype

```
struct _USBH_URB {
  USBH_HEADER Header;
  union {
    USBH_CONTROL_REQUEST   ControlRequest;
    USBH_BULK_INT_REQUEST  BulkIntRequest;
    USBH_ENDPOINT_REQUEST  EndpointRequest;
    USBH_SET_INTERFACE     SetInterface;
    USBH_SET_POWER_STATE   SetPowerState;
  } Request;
};
```

| Member | Description |
|---|---|
| Header | Contains the URB header of type USBH_HEADER. The most important parameters are the function code and the callback function. |
| Request | A union that contains information depending on the specific request of the USBH_HEADER. See description of the individual sub structures. |

## 5.2.14 SEGGER_CACHE_CONFIG

### Description

Used to pass cache configuration and callback function pointers to the stack.

### Prototype

```
typedef struct {
  int  CacheLineSize;
  void (*pfDMB)       (void);
  void (*pfClean)     (void *p, unsigned NumBytes);
  void (*pfInvalidate)(void *p, unsigned NumBytes);
} SEGGER_CACHE_CONFIG;
```

| Member | Description |
|---|---|
| CacheLineSize | Length of one cache line of the CPU.<br>= 0: No Cache.<br>> 0: Cache line size in bytes.<br>Most Systems such as ARM9 use a 32 bytes cache line size. |
| pfDMB | Pointer to a callback function that executes a DMB (Data Memory Barrier) instruction to make sure all memory operations are completed. Can be NULL. |
| pfClean | Pointer to a callback function that executes a clean operation on cached memory. Can be NULL. |
| pfInvalidate | Pointer to a callback function that executes an invalidate operation on cached memory. Can be NULL. |

### Additional information

For further information about how this structure is used please refer to *USBH_SetCacheConfig* on page 77.

## 5.3 Enumerations

The table below lists the available enumerations.

| Structure | Description |
|---|---|
| USBH_DEVICE_EVENT | Enum containing 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 | Enum containing operation speed values of a device. |

## 5.3.1 USBH_DEVICE_EVENT

### Description

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

### Type definition

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

### Enumeration constants

| Constant | Description |
|---|---|
| USBH_DEVICE_EVENT_ADD | Indicates that a device was connected to the host and new interface is available. |
| USBH_DEVICE_EVENT_RE-MOVE | Indicates that a device has been removed. |

## 5.3.2 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.

### Type definition

```
typedef enum {
  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_INTERFACE,
  USBH_FUNCTION_SET_POWER_STATE
} USBH_FUNCTION;
```

### Enumeration constants

| Constant | 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 SetAddress and SetInterface can not be submitted by this request. These requests require a special handling in the driver. See 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. |
| USBH_FUNCTION_RESET_DEVICE | Sends a USB reset to the device. This removes the device and all its interfaces from the USB stack. The application should abort all pending requests and close all handles to this device. All handles become invalid. The USB stack then starts a new enumeration of the device. All interfaces will get new interface Ids. This request can be part of an error recovery or part of special class protocols like DFU. This function uses only the URB header. |
| 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 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. |

| Constant | Description |
|---|---|
| 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. |
| 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. |

## 5.3.3 USBH_PNP_EVENT

### Description

Is used as a parameter for the PnP notification.

### Type definition

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

### Enumeration constants

| Constant | 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. |

## 5.3.4 USBH_POWER_STATE

### Description

Enumerates the power states of a device. Is used as a member in the USBH_SET_POWER_STATE data structure.

### Type definition

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

### Enumeration constants

| Constant | Description |
|---|---|
| USBH_NORMAL_POWER | The device is switched to normal operation. |
| USBH_SUSPEND | The device is switched to USB suspend mode. |
| USBH_POWER_OFF | The device is powered off. |

## 5.3.5 USBH_SPEED

### Description

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

### Type definition

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

### Enumeration constants

| Constant | Description |
|---|---|
| USBH_SPEED_UNKNOWN | The speed is unknown. |
| USBH_LOW_SPEED | The device operates in low-speed mode. |
| USBH_FULL_SPEED | The device operates in full-speed mode. |
| USBH_HIGH_SPEED | The device operates in high-speed mode. |

# 5.4 Function Types

The table below lists the available function types.

| Type | Description |
|---|---|
| USBH_NOTIFICATION_FUNC | Type of user callback set in USBH_PRINTER_RegisterNotification(), USBH_HID_RegisterNotification(), USBH_CDC_AddNotification(), USBH_FT232_RegisterNotification() and USBH_MTP_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. |
| USBH_ON_SETPORTPOWER_FUNC | Callback set by USBH_SetOnSetPortPower(). |

## 5.4.1 USBH_NOTIFICATION_FUNC

### Description

Type of user callback set in `USBH_PRINTER_RegisterNotification()`, `USBH_HID_RegisterNotification()`, `USBH_CDC_AddNotification()`, `USBH_FT232_RegisterNotification()` and `USBH_MTP_RegisterNotification()`.

### Type definition

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

### Parameters

| Parameter | Description |
|---|---|
| `pContext` | Pointer to a context passed by the user in the call to one of the register 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` | Enum `USBH_DEVICE_EVENT` which gives information about the event that occurred. |

## 5.4.2 USBH_ON_COMPLETION_FUNC

### Description

Is called by the library when an URB request completes.

### Type definition

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

### Parameters

| Parameter | Description |
|---|---|
| pUrb | Contains the URB that was completed. |

### Additional information

Is called in the context of the USBH_Task() or USBH_ISRTask().

## 5.4.3 USBH_ON_ENUM_ERROR_FUNC

### Description

Is called by the library if an error occurs at enumeration stage.

### Type definition

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

### Parameters

| Parameter | Description |
| --- | --- |
| pContext | Is a user defined pointer that was passed to USBH_RegisterEnumErrorNotification(). |
| pEnumError | Pointer to a structure containing information about the error occurred. This structure is temporary and must not be accessed after the functions returns. |

### Additional information

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.

## 5.4.4 USBH_ON_PNP_EVENT_FUNC

### Description

Is called by the library if a PnP event occurs and if a PnP notification was registered.

### Type definition

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

### Parameters

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

### Additional information

Is called in the context of USBH_Task() function. In the context of this function all other API functions of the USB host stack 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.

## 5.4.5 USBH_ON_SETPORTPOWER_FUNC

### Description

Callback set by `USBH_SetOnSetPortPower()`. Is called when port power changes.

### Type definition

```
typedef void (USBH_ON_SETPORTPOWER_FUNC)(U32 HostControllerIndex,
                                         U8  Port,
                                         U8  PowerOn);
```

### Parameters

| Parameter | Description |
|---|---|
| `HostControllerIndex` | Index of the host controller. This corresponds to the return value of the respective USBH_<DriverName>_Add call. |
| `Port` | 1-based port index. |
| `PowerOn` | • 0 - power off<br>• 1 - power on |

# 5.5 USBH_STATUS

### Description

Status codes returned by most of the API functions.

### Type definition

```
typedef enum {
  USBH_STATUS_SUCCESS,
  USBH_STATUS_CRC,
  USBH_STATUS_BITSTUFFING,
  USBH_STATUS_DATATOGGLE,
  USBH_STATUS_STALL,
  USBH_STATUS_NOTRESPONDING,
  USBH_STATUS_PID_CHECK,
  USBH_STATUS_UNEXPECTED_PID,
  USBH_STATUS_DATA_OVERRUN,
  USBH_STATUS_DATA_UNDERRUN,
  USBH_STATUS_XFER_SIZE,
  USBH_STATUS_DMA_ERROR,
  USBH_STATUS_BUFFER_OVERRUN,
  USBH_STATUS_BUFFER_UNDERRUN,
  USBH_STATUS_FRAME_ERROR,
  USBH_STATUS_CHANNEL_NAK,
  USBH_STATUS_ERROR,
  USBH_STATUS_INVALID_PARAM,
  USBH_STATUS_PENDING,
  USBH_STATUS_DEVICE_REMOVED,
  USBH_STATUS_CANCELED,
  USBH_STATUS_BUSY,
  USBH_STATUS_INVALID_DESCRIPTOR,
  USBH_STATUS_ENDPOINT_HALTED,
  USBH_STATUS_TIMEOUT,
  USBH_STATUS_PORT,
  USBH_STATUS_INVALID_HANDLE,
  USBH_STATUS_NOT_OPENED,
  USBH_STATUS_ALREADY_ADDED,
  USBH_STATUS_LENGTH,
  USBH_STATUS_COMMAND_FAILED,
  USBH_STATUS_INTERFACE_PROTOCOL,
  USBH_STATUS_INTERFACE_SUB_CLASS,
  USBH_STATUS_WRITE_PROTECT,
  USBH_STATUS_INTERNAL_BUFFER_NOT_EMPTY,
  USBH_STATUS_MTP_OPERATION_NOT_SUPPORTED,
  USBH_STATUS_MEMORY,
  USBH_STATUS_RESOURCES
} USBH_STATUS;
```

### Enumeration constants

| Constant | Description |
|---|---|
| USBH_STATUS_SUCCESS | Operation successfully completed. |
| USBH_STATUS_CRC | Data packet received from device contained a CRC error. |
| USBH_STATUS_BITSTUFFING | Data packet received from device contained a bit stuffing violation. |
| USBH_STATUS_DATATOGGLE | Data packet received from device had data toggle PID that did not match the expected value. |
| USBH_STATUS_STALL | Endpoint was stalled by the device. |
| USBH_STATUS_NOTRESPONDING | USB device did not respond to the request (did not respond to IN token or did not provide a handshake to the OUT token. |

| Constant | Description |
|---|---|
| USBH_STATUS_PID_CHECK | Check bits on PID from endpoint failed on data PID (IN) or handshake (OUT). |
| USBH_STATUS_UNEXPECTED_PID | Receive PID was not valid when encountered or PID value is not defined. |
| USBH_STATUS_DATA_OVERRUN | The amount of data returned by the device exceeded either the size of the maximum data packet allowed from the endpoint or the remaining buffer size (Babble error). |
| USBH_STATUS_DATA_UNDERRUN | The endpoint returned less than maximum packet size and that amount was not sufficient to fill the specified buffer. |
| USBH_STATUS_XFER_SIZE | Size exceeded the maximum transfer size supported by the driver. |
| USBH_STATUS_DMA_ERROR | Direct memory access error. |
| USBH_STATUS_BUFFER_OVERRUN | During an IN transfer, the host controller received data from the device faster than it could be written to system memory. |
| USBH_STATUS_BUFFER_UNDERRUN | During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with data USB data rate. |
| USBH_STATUS_FRAME_ERROR | An interrupt transfer could not be scheduled within a micro frame. |
| USBH_STATUS_CHANNEL_NAK | Internal use. |
| USBH_STATUS_ERROR | Unspecified error occurred. |
| USBH_STATUS_INVALID_PARAM | An invalid parameter was provided. |
| USBH_STATUS_PENDING | The operation was started asynchronously. |
| USBH_STATUS_DEVICE_REMOVED | The device was detached from the host. |
| USBH_STATUS_CANCELED | The operation was canceled by user request. |
| USBH_STATUS_BUSY | The endpoint, interface or device has pending requests and therefore the operation can not be executed. |
| USBH_STATUS_INVALID_DESCRIPTOR | A device provided an invalid descriptor. |
| USBH_STATUS_ENDPOINT_HALTED | The endpoint has been halted. A pipe will be halted when a data transmission error (CRC, bit stuff, DATA toggle) occurs. |
| USBH_STATUS_TIMEOUT | The operation was aborted due to a timeout. |
| USBH_STATUS_PORT | Operation on a USB port failed. |
| USBH_STATUS_INVALID_HANDLE | An invalid handle was provided to the function. |
| USBH_STATUS_NOT_OPENED | The device or interface was not opened. |
| USBH_STATUS_ALREADY_ADDED | Item was already been added. |
| USBH_STATUS_LENGTH | The operation detected a length error. |
| USBH_STATUS_COMMAND_FAILED | This error is reported if the MSD command code was sent successfully but the status returned from the device indicates a command error. |
| USBH_STATUS_INTERFACE_PROTOCOL | The used MSD interface protocol is not supported. The interface protocol is defined by the interface descriptor. |

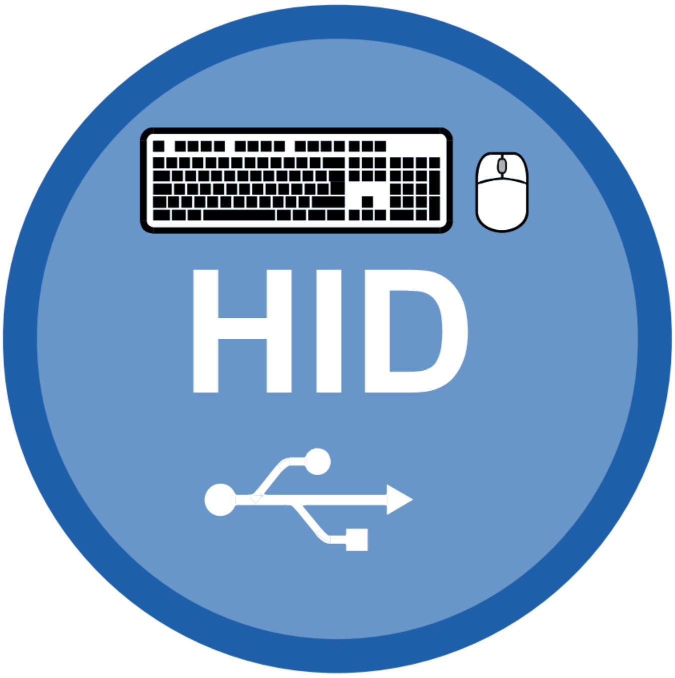
| Constant | Description |
|---|---|
| USBH_STATUS_INTERFACE_SUB_CLASS | The used MSD interface sub class is not supported. The interface sub class is defined by the interface descriptor. |
| USBH_STATUS_WRITE_PROTECT | The MSD medium is write protected. |
| USBH_STATUS_INTERNAL_BUFFER_NOT_EMPTY | Internal use. |
| USBH_STATUS_MTP_OPERATION_NOT_SUPPORTED | The requested MTP operation is not supported by the connected device. |
| USBH_STATUS_MEMORY | Memory could not been allocated. |
| USBH_STATUS_RESOURCES | Not enough resources (e.g endpoints, events, handles, …) |

# 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 BASE 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
- Devices using the HID protocol for data transfer

## 6.1.2 Example code

Example code which is provided in the USBH_HID_Start.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.

| Function | Description |
|---|---|
| USBH_HID_CancelIo() | Cancels any pending 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 stack and unregisters all notification functions. |
| USBH_HID_GetDeviceInfo() | Retrieves information about an opened HID device. |
| USBH_HID_GetNumDevices() | Returns the number of available devices. |
| USBH_HID_GetReport() | Reads a report from a HID device. |
| USBH_HID_GetReportDesc() | Returns the data of a report descriptor in raw form. |
| USBH_HID_Init() | Initializes and registers the HID device driver with emUSB-Host. |
| USBH_HID_Open() | Opens a device given by an index. |
| USBH_HID_RegisterNotification() | Registers a notification callback in order to inform user about adding or removing a device. |
| USBH_HID_SetOnKeyboardStateChange() | Sets a callback to be called in case of keyboard events. |
| USBH_HID_SetOnMouseStateChange() | Sets a callback to be called in case of mouse events. |
| USBH_HID_SetOnGenericEvent() | Sets a callback to be called in case of generic HID events. |
| USBH_HID_SetReport() | Sends a report to a HID device. |
| USBH_HID_SetReportEx() | Sends a report to a HID device. |
| USBH_HID_SetIndicators() | Sets the indicators (usually LEDs) on a keyboard. |
| USBH_HID_GetIndicators() | Retrieves the indicator (LED) status. |
| USBH_HID_ConfigureAllowLEDUpdate() | Sets whether the keyboard LED should be updated or not. |

### 6.2.0.1 USBH_HID_CancelIo()

#### Description

Cancels any pending read/write operation.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the HID device. |

#### Return value

USBH_STATUS_SUCCESS : Operation successfully canceled. Any other value means error.

## 6.2.0.2 USBH_HID_Close()

### Description

Closes a handle to opened HID device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### 6.2.0.3 USBH_HID_Exit()

#### Description

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

#### Prototype

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

### 6.2.0.4 USBH_HID_GetDeviceInfo()

#### Description

Retrieves information about an opened 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 HID device. |
| pDevInfo | Pointer to a USBH_HID_DEVICE_INFO buffer. |

#### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### 6.2.0.5 USBH_HID_GetNumDevices()

#### Description

Returns the number of available devices. It also retrieves the information about a device.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| pDevInfo | Pointer to an array of USBH_HID_DEVICE_INFO structures. |
| NumItems | Number of items that pDevInfo can hold. |

#### Return value

Number of devices available.

# 6.2.0.6 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,
                               USBH_HID_RW_CONTEXT * pRWContext);
```

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an opened HID device. |
| pBuffer | Pointer to a buffer to read. |
| BufferSize | Size of the buffer. |
| pfFunc | [Optional] Callback function of type USBH_HID_USER_FUNC invoked when the read operation finishes (asynchronous operation). It can be the NULL pointer, the function is executed synchronously. |
| pRWContext | [Optional] Pointer to a USBH_HID_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the callback function (pfFunc). If pfFunc ≠ NULL, this parameter is required. If pfFunc = NULL, only the member pRWContext->NumBytesTransferred is set by the function. |

## Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Success on synchronous operation (pfFunc = NULL). |
| USBH_STATUS_PENDING | Request was submitted successfully and the application is informed via callback (pfFunc ≠ NULL). Any other value means error. |

## 6.2.0.7 USBH_HID_GetReportDesc()

### Description

Returns the data of a report descriptor in raw form.

### Prototype

```
USBH_STATUS USBH_HID_GetReportDesc(      USBH_HID_HANDLE    hDevice,
                                   const U8              ** ppReportDescriptor,
                                         unsigned          * pNumBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an opened device. |
| ppReportDescriptor | Returns a pointer to the report descriptor which is stored in an internal data structure of the USB stack. The report descriptor must not be changed. The pointer becomes invalid after the device is closed. |
| pNumBytes | Returns the size of the report descriptor in bytes. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 6.2.0.8 USBH_HID_Init()

### Description

Initializes and registers the HID device driver with emUSB-Host.

### Prototype

```
U8 USBH_HID_Init(void);
```

### Return value

| | |
|---|---|
| 1 | Success. |
| 0 | Could not register HID device driver. |

# 6.2.0.9 USBH_HID_Open()

## Description

Opens a device given by an index.

## Prototype

```
USBH_HID_HANDLE USBH_HID_Open(unsigned Index);
```

## Parameters

| Parameter | Description |
|---|---|
| Index | Device index. |

## Return value

$\neq 0$ Handle to a HID device.
$= 0$ Device not available.

## Additional information

The index of a new connected device is provided to the callback function registered with `USBH_HID_RegisterNotification()`.

## 6.2.0.10 USBH_HID_RegisterNotification()

### Description

Registers a notification callback in order to inform user about adding or removing a device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pfNotification | 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 callback function. Any value the application chooses is permitted, including NULL. |

# 6.2.0.11 USBH_HID_SetOnKeyboardStateChange()

## Description

Sets a callback to be called in case of keyboard events.

## Prototype

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

## Parameters

| Parameter | Description |
|---|---|
| pfOnChange | Callback that shall be called when a keyboard change notification is available. |

## 6.2.0.12 USBH_HID_SetOnMouseStateChange()

### Description

Sets a callback to be called in case of mouse events.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pfOnChange | Callback that shall be called when a mouse change notification is available. |

## 6.2.0.13 USBH_HID_SetOnGenericEvent()

### Description

Sets a callback to be called in case of generic HID events.

### Prototype

```
void USBH_HID_SetOnGenericEvent(      U32                          NumUsages,
                                const U32                        * pUsages,
                                      USBH_HID_ON_GENERIC_FUNC * pfOnEvent);
```

### Parameters

| Parameter | Description |
|---|---|
| NumUsages | Number of usage codes provided by the caller. |
| pUsages | List of usage codes of fields from the report to be monitored. Each usage code must contain the Usage Page in the high order 16 bits and the Usage ID in the the low order 16 bits. pUsages must point to a static memory area that remains valid until the USBH_HID module is shut down. |
| pfOnEvent | Callback that shall be called when a report is received that contains at least one field with usage code from the list. |

## 6.2.0.14 USBH_HID_SetReport()

### Description

Sends a report to a HID device. This function assumes report IDs are not used.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an opened HID device. |
| pBuffer | Pointer to a buffer containing the data to be sent. In case the device has more than one report descriptor the first byte inside the buffer must contain a valid ID matching one of the report descriptors. |
| BufferSize | Size of the buffer. |
| pfFunc | [Optional] Callback function of type USBH_HID_USER_FUNC invoked when the send operation finishes. It can be the NULL pointer. |
| pRWContext | [Optional] Pointer to a USBH_HID_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function. |

### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Success. |
| 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. |

## 6.2.0.15 USBH_HID_SetReportEx()

### Description

Sends a report to a HID device. Optionally sends out a report ID.

### Prototype

```
USBH_STATUS USBH_HID_SetReportEx(      USBH_HID_HANDLE       hDevice,
                                 const U8                  * pBuffer,
                                       U32                   BufferSize,
                                       USBH_HID_USER_FUNC  * pfFunc,
                                       USBH_HID_RW_CONTEXT * pRWContext,
                                       U8                    UseReportIDs);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an opened HID device. |
| pBuffer | Pointer to a buffer containing the data to be sent. In case the device has more than one report descriptor the first byte inside the buffer must contain a valid ID matching one of the report descriptors. |
| BufferSize | Size of the buffer. |
| pfFunc | [Optional] Callback function of type USBH_HID_USER_FUNC invoked when the send operation finishes. It can be the NULL pointer. |
| pRWContext | [Optional] Pointer to a USBH_HID_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function. |
| UseReportIDs | Flag which enables or disables report ID usage. 1 - use report IDs, 0 - do not use report IDs. If report IDs are being used the report ID should be the first byte in the buffer pointed to by pBuffer. |

### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Success. |
| 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. |

## 6.2.0.16 USBH_HID_SetIndicators()

### Description

Sets the indicators (usually LEDs) on a keyboard.

### Prototype

```
USBH_STATUS USBH_HID_SetIndicators(USBH_HID_HANDLE hDevice,
                                   U8              IndicatorMask);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| IndicatorMask | Binary mask of the following items<br>USBH_HID_IND_NUM_LOCK<br>USBH_HID_IND_CAPS_LOCK<br>USBH_HID_IND_SCROLL_LOCK<br>USBH_HID_IND_COMPOSE<br>USBH_HID_IND_KANA<br>USBH_HID_IND_SHIFT |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 6.2.0.17 USBH_HID_GetIndicators()

## Description

Retrieves the indicator (LED) status.

## Prototype

```
USBH_STATUS USBH_HID_GetIndicators(USBH_HID_HANDLE   hDevice,
                                   U8              * pIndicatorMask);
```

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pIndicatorMask | Binary mask of the following items<br>USBH_HID_IND_NUM_LOCK<br>USBH_HID_IND_CAPS_LOCK<br>USBH_HID_IND_SCROLL_LOCK<br>USBH_HID_IND_COMPOSE<br>USBH_HID_IND_KANA<br>USBH_HID_IND_SHIFT |

## Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 6.2.0.18 USBH_HID_ConfigureAllowLEDUpdate()

### Description

Sets whether the keyboard LED should be updated or not. (Default is yes).

### Prototype

```
void USBH_HID_ConfigureAllowLEDUpdate(unsigned AllowLEDUpdate);
```

### Parameters

| Parameter | Description |
|---|---|
| AllowLEDUpdate | • 0 - Disable LED Update.<br>• 1 - Allow LED Update. |

## 6.3 Data structures

This chapter describes the emUSB-Host HID API structures.

| Structure | Description |
|---|---|
| USBH_HID_DEVICE_INFO | Structure containing information about a HID device. |
| USBH_HID_KEYBOARD_DATA | Structure containing information about a keyboard event. |
| USBH_HID_MOUSE_DATA | Structure containing information about a mouse event. |
| USBH_HID_GENERIC_DATA | Structure containing information from a HID report event. |
| USBH_HID_REPORT_INFO | Structure containing information about a HID report. |
| USBH_HID_RW_CONTEXT | Contains information about a completed, asynchronous transfers. |

# 6.3.0.1 USBH_HID_DEVICE_INFO

## Description

Structure containing information about a HID device.

## Type definition

```
typedef struct {
  U16                   InputReportSize;
  U16                   OutputReportSize;
  U16                   ProductId;
  U16                   VendorId;
  unsigned              DevIndex;
  USBH_INTERFACE_ID     InterfaceID;
  unsigned              NumReportInfos;
  USBH_HID_REPORT_INFO  ReportInfo[];
  U8                    DeviceType;
} USBH_HID_DEVICE_INFO;
```

## Structure members

| Member | Description |
|---|---|
| InputReportSize | = ReportInfo[0].InputReportSize for compatibility. |
| OutputReportSize | = ReportInfo[0].OutputReportSize for compatibility. |
| ProductId | The Product ID of the device. |
| VendorId | The Vendor ID of the device. |
| DevIndex | Device index. |
| InterfaceID | Interface ID of the HID device. |
| NumReportInfos | Number of entries in ReportInfo. |
| ReportInfo | Size and Report Ids of all reports of the interface. |
| DeviceType | Device type.<br>• USBH_HID_VENDOR - Vendor device<br>• USBH_HID_MOUSE - Mouse device<br>• USBH_HID_KEYBOARD - Keyboard device |

### 6.3.0.2 USBH_HID_KEYBOARD_DATA

#### Description

Structure containing information about a keyboard event.

#### Type definition

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

#### Structure members

| Member | Description |
|---|---|
| Code | Contains the keycode. |
| Value | Keyboard state info. Refer to sample code for more information. |
| InterfaceID | ID of the interface that caused the event. |

### 6.3.0.3 USBH_HID_MOUSE_DATA

#### Description

Structure containing information about a mouse event.

#### Type definition

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

#### Structure 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,<br>• bit 0 corresponds to the left mouse button<br>• bit 1 corresponds to the right mouse button<br>• bit 2 corresponds to the middle mouse button. |
| InterfaceID | ID of the interface that caused the event. |

# 6.3.0.4 USBH_HID_GENERIC_DATA

## Description

Structure containing information from a HID report event.

## Type definition

```
typedef struct {
  U32               Usage;
  USBH_ANY_SIGNED   Value;
  U8                Valid;
  U8                Signed;
  U8                ReportID;
  USBH_ANY_SIGNED   LogicalMin;
  USBH_ANY_SIGNED   LogicalMax;
  USBH_ANY_SIGNED   PhysicalMin;
  USBH_ANY_SIGNED   PhysicalMax;
  U8                PhySigned;
  U8                NumBits;
  U16               BitPosStart;
} USBH_HID_GENERIC_DATA;
```

## Structure members

| Member | Description |
|---|---|
| Usage | HID usage code. Copied from the array given to USBH_HID_SetOnGenericEvent(). Set to 0, if the usage code was not found in any report descriptor. |
| Value | Value of the field extracted from the report. |
| Valid | = 1 if Value field contains valid value. |
| Signed | = 1 if Value is signed, = 0 if unsigned. |
| ReportID | ID of the report containing the field. |
| LogicalMin | Logical minimum from report descriptor. Contains signed value, if Signed = 1, unsigned value otherwise. |
| LogicalMax | Logical maximum from report descriptor. Contains signed value, if Signed = 1, unsigned value otherwise. |
| PhysicalMin | Physical minimum from report descriptor. Contains signed value, if PhySigned = 1, unsigned value otherwise. |
| PhysicalMax | Physical maximum from report descriptor. Contains signed value, if PhySigned = 1, unsigned value otherwise. |
| PhySigned | = 1 if PhysicalMin / PhysicalMax are signed, = 0 if unsigned. |
| NumBits | Internal use. |
| BitPosStart | Internal use. |

## 6.3.0.5 USBH_HID_REPORT_INFO

### Description

Structure containing information about a HID report.

### Type definition

```
typedef struct {
  U8   ReportId;
  U16  InputReportSize;
  U16  OutputReportSize;
} USBH_HID_REPORT_INFO;
```

### Structure members

| Member | Description |
|---|---|
| ReportId | Report Id |
| InputReportSize | Size of input report in bytes. |
| OutputReportSize | Size of output report in bytes. |

### 6.3.0.6 USBH_HID_RW_CONTEXT

#### Description

Contains information about a completed, asynchronous transfers. Is passed to the USBH_HID_ON_COMPLETE_FUNC user callback when using asynchronous write and read. When this structure is passed to USBH_HID_GetReport() or USBH_HID_SetReport() its member need not to be initialized.

#### Type definition

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

#### Structure members

| Member | Description |
|---|---|
| pUserContext | Pointer to a user context. Can be arbitrarily used by the application. |
| Status | Result status of the asynchronous transfer. |
| NumBytesTransferred | Number of bytes transferred. |
| pUserBuffer | Pointer to the buffer provided to USBH_HID_GetReport() or USBH_HID_SetReport(). |
| UserBufferSize | Size of the buffer as provided to USBH_HID_GetReport() or USBH_HID_SetReport(). |

# 6.4 Function Types

This chapter describes the emUSB-Host HID API function types.

| Type | Description |
|---|---|
| USBH_HID_ON_KEYBOARD_FUNC | Function called on every keyboard event. |
| USBH_HID_ON_MOUSE_FUNC | Function called on every mouse event. |
| USBH_HID_ON_GENERIC_FUNC | Function called on every generic HID event. |
| USBH_HID_USER_FUNC | Function called on completion of USBH_HID_GetReport() or USBH_HID_SetReport(). |

### 6.4.0.1 USBH_HID_ON_KEYBOARD_FUNC

#### Description

Function called on every keyboard event.

#### Type definition

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

#### Parameters

| Parameter | Description |
|---|---|
| pKeyData | Pointer to a USBH_HID_KEYBOARD_DATA structure. |

### 6.4.0.2 USBH_HID_ON_MOUSE_FUNC

#### Description

Function called on every mouse event.

#### Type definition

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

#### Parameters

| Parameter | Description |
|---|---|
| pMouseData | Pointer to a USBH_HID_MOUSE_DATA structure. |

### 6.4.0.3 USBH_HID_ON_GENERIC_FUNC

#### Description

Function called on every generic HID event.

#### Type definition

```
typedef void (USBH_HID_ON_GENERIC_FUNC)
                                        (       USBH_INTERFACE_ID       InterfaceID,
                                                unsigned                NumGenericInfos,
                                          const USBH_HID_GENERIC_DATA * pGenericData);
```

#### Parameters

| Parameter | Description |
|---|---|
| InterfaceID | Interface ID of the HID device that generated the event. |
| NumGenericInfos | Number of USBH_HID_GENERIC_DATA structures provided. |
| pGenericData | Pointer to an array of USBH_HID_GENERIC_DATA structures. |

## 6.4.0.4 USBH_HID_USER_FUNC

### Description

Function called on completion of `USBH_HID_GetReport()` or `USBH_HID_SetReport()`.

### Type definition

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

### Parameters

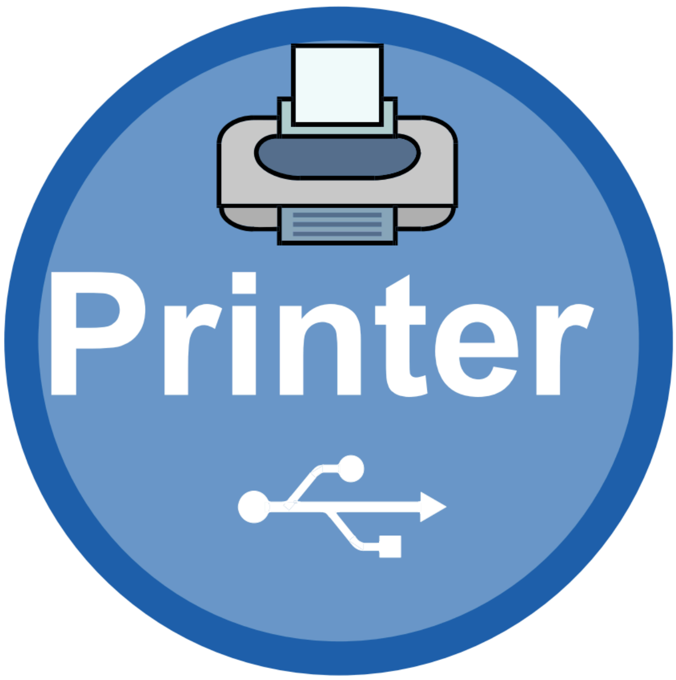
| Parameter | Description |
|---|---|
| `pContext` | Pointer to a `USBH_HID_RW_CONTEXT` structure. |

# 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 extension 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 `USBH_Printer_Start.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.

| Function | Description |
|---|---|
| USBH_PRINTER_Close() | Closes a handle to an opened printer. |
| USBH_PRINTER_ConfigureTimeout() | Sets up the default timeout the host waits until the data transfer will be aborted. |
| USBH_PRINTER_ExecSoftReset() | Flushes all send and receive buffers. |
| USBH_PRINTER_Exit() | Exit from application. |
| USBH_PRINTER_GetDeviceId() | Ask the USB printer to send the IEEE.1284 ID string. |
| USBH_PRINTER_GetNumDevices() | Returns the number of available devices. |
| USBH_PRINTER_GetPortStatus() | Returns the status of printer. |
| USBH_PRINTER_Init() | Initialize the Printer device class driver. |
| USBH_PRINTER_Open() | Opens a handle to a printer. |
| USBH_PRINTER_OpenByIndex() | Opens a device given by an 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. |

#### 7.2.0.1 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. |

**Return value**

| | |
|---|---|
| USBH_STATUS_SUCCESS | Handle closed. |
| USBH_STATUS_ERROR | Invalid handle. |

## 7.2.0.2 USBH_PRINTER_ConfigureTimeout()

### Description

Sets up the default timeout the host waits until the data transfer will be aborted.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| Timeout | Timeout given in ms. |

### 7.2.0.3 USBH_PRINTER_ExecSoftReset()

#### Description

Flushes all send and receive buffers.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened printer. |

#### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Reset executed. |
| USBH_STATUS_ERROR | An error occurred. |

## 7.2.0.4 USBH_PRINTER_Exit()

### Description

Exit from application. Application has to wait until that all URB requests completed before this function is called!

### Prototype

```
void USBH_PRINTER_Exit(void);
```

## 7.2.0.5 USBH_PRINTER_GetDeviceId()

### Description

Ask the USB printer to send the IEEE.1284 ID string.

### Prototype

```
USBH_STATUS USBH_PRINTER_GetDeviceId(USBH_PRINTER_HANDLE   hDevice,
                                     U8                  * pData,
                                     unsigned              NumBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened printer device. |
| pData | Pointer to a caller allocated buffer. |
| NumBytes | Number of bytes allocated for the read buffer. |

### Return value

USBH_STATUS_SUCCESS : Device ID read. Any other status : An error occurred.

### 7.2.0.6 USBH_PRINTER_GetNumDevices()

#### Description

Returns the number of available devices.

#### Prototype

```
int USBH_PRINTER_GetNumDevices(void);
```

#### Return value

Number of devices available

### 7.2.0.7 USBH_PRINTER_GetPortStatus()

#### Description

Returns the status of printer.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened printer. |
| pStatus | Pointer to a caller allocated variable. |

#### Return value

= USBH_STATUS_SUCCESS      Status retrieved successfully.
≠ USBH_STATUS_SUCCESS      An error occurred.

#### Additional information

The returned status is to be interpreted as follows:

| Bit(s) | Fields | Explanations |
|---|---|---|
| 7 .. 6 | Reserved | Reserved for future use; device shall return these bits set 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 set to zero. |

### 7.2.0.8 USBH_PRINTER_Init()

#### Description

Initialize the Printer device class driver.

#### Prototype

```
U8 USBH_PRINTER_Init(void);
```

#### Return value

| | |
|---|---|
| 1 | Success |
| 0 | Could not register class device driver |

## 7.2.0.9 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 | Pointer to a name of the device eg. prt001 for device 0. |

### Return value

≠ 0 Handle to a printing device
= 0 Device not available or error occurred.

### Additional information

It is recommended to use `USBH_PRINTER_OpenByIndex()`. It is slightly faster.

## 7.2.0.10 USBH_PRINTER_OpenByIndex()

### Description

Opens a device given by an index.

### Prototype

```
USBH_PRINTER_HANDLE USBH_PRINTER_OpenByIndex(unsigned Index);
```

### Parameters

| Parameter | Description |
| --- | --- |
| Index | Device index. |

### Return value

≠ 0 Handle to a printer device.
= 0 Device not available.

### Additional information

The index of a new connected device is provided to the callback function registered with USBH_PRINTER_RegisterNotification().

## 7.2.0.11 USBH_PRINTER_Read()

### Description

Receives data from a printer.

### Prototype

```
USBH_STATUS USBH_PRINTER_Read(USBH_PRINTER_HANDLE   hDevice,
                              U8                  * pData,
                              unsigned              NumBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened printer. |
| pData | Pointer to a caller allocated buffer. |
| NumBytes | Size of the receive buffer in bytes. |

### Return value

= USBH_STATUS_SUCCESS     Data received.
≠ USBH_STATUS_SUCCESS     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.

## 7.2.0.12 USBH_PRINTER_RegisterNotification()

### Description

Registers a notification for the printer connect/disconnect events.

### Prototype

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

### Parameters

| 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 call-back function. |

### Additional information

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

## 7.2.0.13 USBH_PRINTER_Write()

### Description

Sends data to a printer.

### Prototype

```
USBH_STATUS USBH_PRINTER_Write(      USBH_PRINTER_HANDLE   hDevice,
                               const U8                  * pData,
                                     unsigned              NumBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened printer. |
| pData | Pointer to data to be sent. |
| NumBytes | Number of bytes to send. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Data sent. |
| ≠ USBH_STATUS_SUCCESS | 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, Adobe Postscript or Microsoft Windows Printing System (WPS).

# Chapter 8

# Mass Storage Device (MSD) class

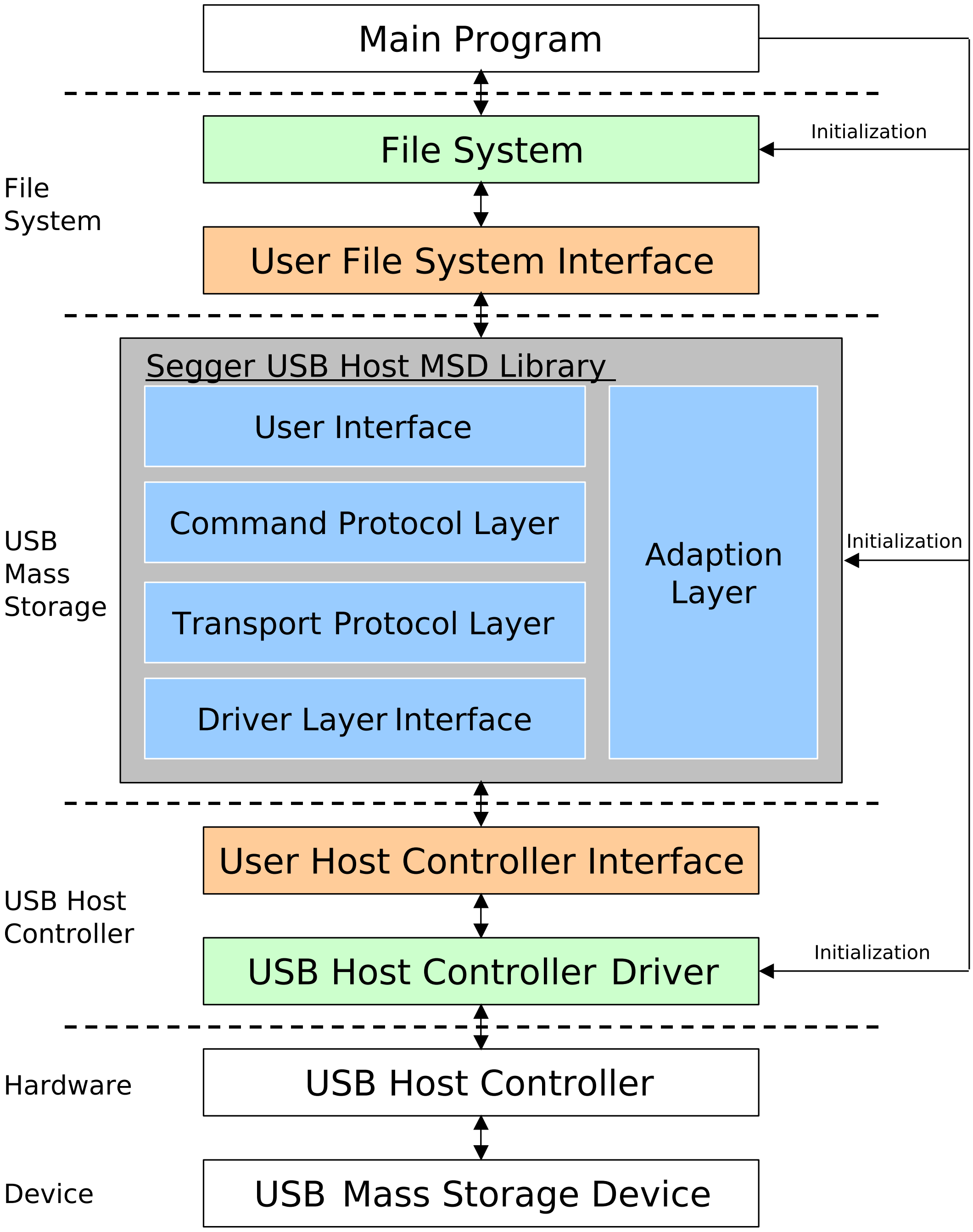
This chapter describes the emUSB-Host Mass storage device class driver and its usage. The MSD class is part of the BASE package. The MSD class code is only linked in if registered by the application program.

# 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 class driver.

## 8.1.1 Overview

A mass storage device connected to the emUSB-Host is added to the file system as a 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 Requirements

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

## 8.1.4 Example code

Example code which is provided in the file `USBH_MSD_Start.c`. The example shows the capacity of the connected device, shows files in the root directory and creates and writes to a file.

## 8.1.5 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) |

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.

| Function | Description |
|---|---|
| USBH_MSD_Exit() | Releases all resources, closes all handles to the USB bus driver and un-register all notification functions. |
| USBH_MSD_GetStatus() | Checks the Status of a device. |
| USBH_MSD_GetUnits() | Returns available units for a device. |
| USBH_MSD_GetUnitInfo() | Returns basic information about the logical unit (LUN). |
| USBH_MSD_GetPortInfo() | Retrieves the port information about a USB MSC device using a unit ID. |
| USBH_MSD_Init() | Initializes the USB Mass Storage Class Driver. |
| USBH_MSD_ReadSectors() | Reads sectors from a USB Mass Storage device. |
| USBH_MSD_WriteSectors() | Writes sectors to a USB Mass Storage device. |
| USBH_MSD_UseAheadCache() | Enables the read-ahead-cache functionality. |
| USBH_MSD_SetAheadBuffer() | Sets a user provided buffer for the read-ahead-cache functionality. |

# 8.2.0.1 USBH_MSD_Exit()

## Description

Releases all resources, closes all handles to the USB bus driver and un-register all notification functions. Has to be called if the application is closed before the `USBH_Exit` is called.

## Prototype

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

# 8.2.0.2 USBH_MSD_GetStatus()

## Description

Checks the Status of a device. Therefore it calls `USBH_MSD_GetUnit` to test if the device is still connected and if a logical unit is assigned.

## Prototype

```
USBH_STATUS USBH_MSD_GetStatus(U8 Unit);
```

## Parameters

| Parameter | Description |
|---|---|
| `Unit` | 0-based `Unit` Id. See `USBH_MSD_GetUnits()`. |

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Device is ready for operation. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### 8.2.0.3 USBH_MSD_GetUnits()

#### Description

Returns available units for a device.

#### Prototype

```
USBH_STATUS USBH_MSD_GetUnits(U8    DevIndex,
                              U32 * pUnitMask);
```

#### Parameters

| Parameter | Description |
|---|---|
| DevIndex | Index of the MSD device returned by USBH_MSD_LUN_NOTIFICATION_FUNC. |
| pUnitMask | out Pointer to a U32 variable which will receive the LUN mask. |

#### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Device is ready for operation. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

#### Additional information

The mask corresponds to the unit IDs. E.g. a mask of 0x0000000C means unit ID 2 and unit ID 3 are available for the device.

## 8.2.0.4 USBH_MSD_GetUnitInfo()

### Description

Returns basic information about the logical unit (LUN).

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| Unit | 0-based Unit Id. See USBH_MSD_GetUnits(). |
| pInfo | out Pointer to a caller provided structure of type USBH_MSD_UNIT_INFO. It receives the information about the LUN in case of success. |

### Return value

= USBH_STATUS_SUCCESS Device is ready for operation.
≠ USBH_STATUS_SUCCESS An error occurred.

## 8.2.0.5 USBH_MSD_GetPortInfo()

### Description

Retrieves the port information about a USB MSC device using a unit ID.

### Prototype

```
USBH_STATUS USBH_MSD_GetPortInfo(U8                 Unit,
                                 USBH_PORT_INFO * pPortInfo);
```

### Parameters

| Parameter | Description |
| --- | --- |
| Unit | 0-based Unit Id. See USBH_MSD_GetUnits(). |
| pPortInfo | out Pointer to a caller provided structure of type USBH_PORT_INFO. It receives the information about the LUN in case of success. |

### Return value

= USBH_STATUS_SUCCESS Success, pPortInfo contains valid port information.
≠ USBH_STATUS_SUCCESS An error occurred.

### 8.2.0.6 USBH_MSD_Init()

#### Description

Initializes the USB Mass Storage Class Driver.

#### 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 to pfLunNotification. |

#### Return value

1 Success.
0 Initialization failed.

#### Additional information

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

#### 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:
    USBH_Logf_Application("**** Device added\n");
    _MSDReady = 1;
    _CurrentDevIndex = DevIndex;
    break;
  case USBH_MSD_EVENT_REMOVE:
    USBH_Logf_Application("**** Device removed\n");
    _MSDReady = 0;
    _CurrentDevIndex = 0xff;
    break;
  default:;   // Should never happen
  }
}

<...>
USBH_MSD_Init(_cbOnAddRemoveDevice, NULL);
<...>
```

### 8.2.0.7 USBH_MSD_ReadSectors()

#### Description

Reads sectors from a USB Mass Storage device. To read file and folders use the file system functions. This function allows to read sectors raw.

#### Prototype

```
USBH_STATUS USBH_MSD_ReadSectors(U8    Unit,
                                 U32   SectorAddress,
                                 U32   NumSectors,
                                 U8  * pBuffer);
```

#### Parameters

| Parameter | Description |
|---|---|
| Unit | 0-based Unit Id. See USBH_MSD_GetUnits(). |
| SectorAddress | Index of the first sector to read. The first sector has the index 0. |
| NumSectors | Number of sectors to read. |
| pBuffer | Pointer to a caller allocated buffer. |

#### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Sectors successfully read. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 8.2.0.8 USBH_MSD_WriteSectors()

### Description

Writes sectors to a USB Mass Storage device. To write files and folders use the file system functions. This function allows to write sectors raw.

### Prototype

```
USBH_STATUS USBH_MSD_WriteSectors(      U8    Unit,
                                        U32   SectorAddress,
                                        U32   NumSectors,
                                  const U8  * pBuffer);
```

### Parameters

| Parameter | Description |
|---|---|
| Unit | 0-based Unit Id. See USBH_MSD_GetUnits(). |
| SectorAddress | Index of the first sector to write. The first sector has the index 0. |
| NumSectors | Number of sectors to write. |
| pBuffer | Pointer to the data. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Sectors successfully written. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 8.2.0.9 USBH_MSD_UseAheadCache()

### Description

Enables the read-ahead-cache functionality.

### Prototype

```
void USBH_MSD_UseAheadCache(int OnOff);
```

### Parameters

| Parameter | Description |
|---|---|
| OnOff | 1 : on, 0 - off. |

### Additional information

The read-ahead-cache is a functionality which makes sure that read accesses to an MSD will always read a minimal amount of sectors (normally at least four). The rest of the sectors which have not been requested directly will be stored in a cache and subsequent reads will be supplied with data from the cache instead of the actual device.

This functionality is mainly used as a workaround for certain MSD devices which crash when single sectors are being read directly from the device too often. Enabling the cache will cause a slight drop in performance, but will make sure that all MSD devices which are affected by the aforementioned issue do not crash. Unless `USBH_MSD_SetAheadBuffer()` was used before calling this function with a "1" as parameter the function will try to allocate a buffer for eight sectors (4096 bytes) from the emUSB-Host memory pool.

## 8.2.0.10 USBH_MSD_SetAheadBuffer()

### Description

Sets a user provided buffer for the read-ahead-cache functionality.

### Prototype

```
void USBH_MSD_SetAheadBuffer(const USBH_MSD_AHEAD_BUFFER * pAheadBuf);
```

### Parameters

| Parameter | Description |
|---|---|
| pAheadBuf | Pointer to a USBH_MSD_AHEAD_BUFFER structure which holds the buffer information. |

### Additional information

This function has to be called before enabling the read-ahead-cache with USBH_MSD_UseAheadCache(). The buffer should have space for at least four sectors (2048 bytes), but eight sectors (4096 bytes) are suggested for better performance. The buffer size must be a multiple of 512.

# 8.3 Data Structures

This chapter describes the used emUSB-Host MSD API structures.

| Function | Description |
|---|---|
| USBH_MSD_UNIT_INFO | Contains logical unit information. |
| USBH_MSD_AHEAD_BUFFER | Structure describing the read-ahead-cache buffer. |

### 8.3.0.1 USBH_MSD_UNIT_INFO

#### Description

Contains logical unit information.

#### Type definition

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

#### Structure 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. |

## 8.3.0.2 USBH_MSD_AHEAD_BUFFER

### Description

Structure describing the read-ahead-cache buffer.

### Type definition

```
typedef struct {
  U8 * pBuffer;
  U32  Size;
} USBH_MSD_AHEAD_BUFFER;
```

### Structure members

| Member | Description |
| --- | --- |
| pBuffer | Pointer to a buffer. |
| Size | Size of the buffer in bytes. |

## 8.4 Function Types

This chapter describes the used emUSB-Host MSD API function types.

| Type | Description |
|---|---|
| USBH_MSD_LUN_NOTIFICATION_FUNC | This callback function is called when a logical unit is either added or removed. |

### 8.4.0.1 USBH_MSD_LUN_NOTIFICATION_FUNC

#### 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.

#### Type definition

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

#### Parameters

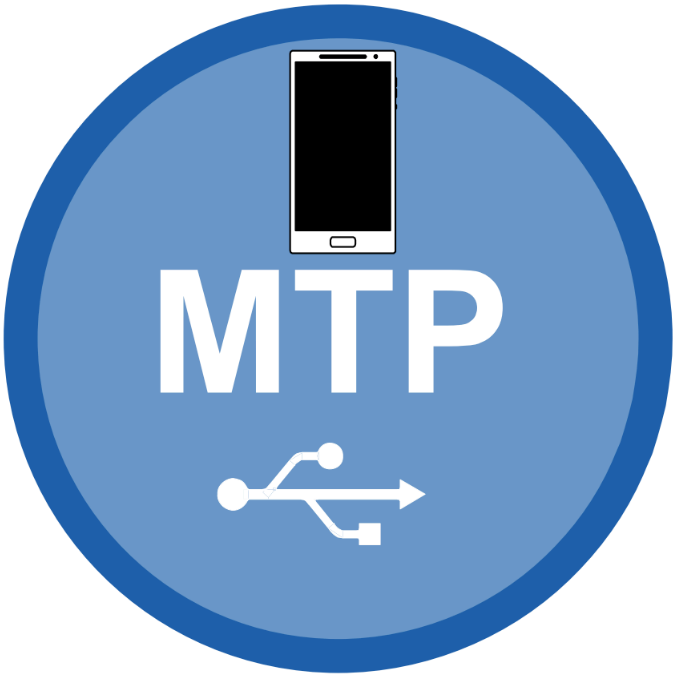
| Parameter | Description |
|---|---|
| `pContext` | Pointer to a context that was set by the user when the `USBH_MSD_Init()` was called. |
| `DevIndex` | Zero based index of the device that was attached or removed. First device has index 0, second one has index 1, etc. |
| `Event` | Gives information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

# Chapter 9

# MTP Device Driver (Add-On)

This chapter describes the optional emUSB-Host add-on “MTP device driver”. It allows communication with MTP USB devices.



# 9.1 Introduction

The MTP driver software component of emUSB-Host allows communication with MTP devices such as Android or Windows smartphones, media players, cameras and so on. A file system is not required to use emUSB-Host MTP. This chapter provides an explanation of the functions available to application developers via the MTP driver software. All the functions and data types of this add-on are prefixed with the “`USBH_MTP_`” text.

## 9.1.1 Overview

An MTP device connected to the emUSB-Host is automatically configured and added to an internal list. If the MTP module has been registered, it is notified via a callback when an MTP device has been added or removed. The driver then can notify the application program when a callback function has been registered via `USBH_MTP_RegisterNotification()`. In order to communicate with such a device, the application has to call `USBH_MTP_Open()`, passing the device index. MTP 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 MTP devices.

## 9.1.3 Example code

An example application which demonstrates the API is provided in the `USBH_MTP_Start.c` file.

## 9.2 API Functions

This chapter describes the emUSB-Host MTP driver API functions.

| Function | Description |
|---|---|
| USBH_MTP_Init() | Initializes and registers the MTP device driver with emUSB-Host. |
| USBH_MTP_Exit() | Unregisters and de-initializes the MTP device driver from emUSB-Host. |
| USBH_MTP_RegisterNotification() | Sets a callback in order to be notified when a device is added or removed. |
| USBH_MTP_Open() | Opens a device using the given index. |
| USBH_MTP_Close() | Closes a handle to an opened device. |
| USBH_MTP_GetDeviceInfo() | Retrieves basic information about the MTP device. |
| USBH_MTP_GetNumStorages() | Retrieves the number of storages the device has. |
| USBH_MTP_Reset() | Executes the MTP reset command on the device. |
| USBH_MTP_SetTimeouts() | Sets timeouts for read and write transactions for a device. |
| USBH_MTP_GetLastErrorCode() | Returns the error code for the last executed operation. |
| USBH_MTP_GetStorageInfo() | Retrieves information about a storage on the device. |
| USBH_MTP_Format() | Formats (deletes all data!) on a device storage. |
| USBH_MTP_GetNumObjects() | Retrieves the number of objects inside a single directory. |
| USBH_MTP_GetObjectList() | Retrieves a list of object IDs from a directory. |
| USBH_MTP_GetObjectInfo() | Retrieves the ObjectInfo dataset for a specific object. |
| USBH_MTP_CreateObject() | Writes a new object onto the device. |
| USBH_MTP_DeleteObject() | Deletes an object from the device. |
| USBH_MTP_Rename() | Changes the name of an object. |
| USBH_MTP_ReadFile() | Reads a file from the device. |
| USBH_MTP_GetDevicePropDesc() | Retrieves the description of a MTP property from the device. |
| USBH_MTP_GetDevicePropValue() | Retrieves the value of a property of a specific Device. |
| USBH_MTP_GetObjectPropsSupported() | Retrieves a list of supported properties for a given object format. |
| USBH_MTP_GetObjectPropDesc() | Retrieves information about an MTP object property used by the device. |
| USBH_MTP_GetObjectPropValue() | Retrieves the value of a property of a specific object. |
| USBH_MTP_SetObjectProperty() | Sets the property of an object to the specified value. |
| USBH_MTP_CheckLock() | Determines whether the device is locked by a pin/password/etc. |

| Function | Description |
| --- | --- |
| USBH_MTP_SetEventCallback() | Sets a callback for MTP events, e.g. |
| USBH_MTP_ConfigEventSupport() | Turns MTP event support on or off. |
| USBH_MTP_GetEventSupport() | Returns the event support configuration, see USBH_MTP_ConfigEventSupport() for details. |

### 9.2.0.1 USBH_MTP_Init()

#### Description

Initializes and registers the MTP device driver with emUSB-Host.

#### Prototype

```
USBH_STATUS USBH_MTP_Init(void);
```

#### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Success. |
| USBH_STATUS_MEMORY | Can not init MTP module, out of memory. |

### 9.2.0.2 USBH_MTP_Exit()

#### Description

Unregisters and de-initializes the MTP device driver from emUSB-Host.

#### Prototype

```
void USBH_MTP_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_MTP_Exit()`. The only exception is `USBH_MTP_Init()`, which would in turn re-init the module and allow further calls.

### 9.2.0.3 USBH_MTP_RegisterNotification()

#### Description

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

#### Prototype

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

#### Parameters

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

#### Additional information

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

## 9.2.0.4 USBH_MTP_Open()

### Description

Opens a device using the given index.

### Prototype

```
USBH_MTP_DEVICE_HANDLE USBH_MTP_Open(U8 Index);
```

### Parameters

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

### Return value

| | |
|---|---|
| ≠ 0 | Handle to the device |
| = 0 | Device not available or removed. |

## 9.2.0.5 USBH_MTP_Close()

### Description

Closes a handle to an opened device.

### Prototype

```
USBH_STATUS USBH_MTP_Close(USBH_MTP_DEVICE_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 9.2.0.6 USBH_MTP_GetDeviceInfo()

### Description

Retrieves basic information about the MTP device.

### Prototype

```
USBH_STATUS USBH_MTP_GetDeviceInfo(USBH_MTP_DEVICE_HANDLE   hDevice,
                                   USBH_MTP_DEVICE_INFO   * pDevInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pDevInfo | out Pointer to a USBH_MTP_DEVICE_INFO structure where the information related to the device will be stored. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 9.2.0.7 USBH_MTP_GetNumStorages()

### Description

Retrieves the number of storages the device has.

### Prototype

```
USBH_STATUS USBH_MTP_GetNumStorages(USBH_MTP_DEVICE_HANDLE   hDevice,
                                    U8                     * pNumStorages);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pNumStorages | out Pointer to a variable where the number of storages reported by the device will be stored. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Additional information

This function may return zero storages when the device is locked. Unfortunately this is not always the case and can not be used as a criteria to check whether a device is locked (e.g. Windows Phones will return the correct number of storages even if they are locked.)

See USBH_MTP_CheckLock() for further information.

# 9.2.0.8 USBH_MTP_Reset()

## Description

Executes the MTP reset command on the device. This command sets the device in the default state. “Default state” can mean different things for different manufacturers. This MTP command is rarely supported by devices. This command will close all sessions on the device side. Therefore the host application should call `USBH_MTP_Close()` after a successful call to this function.

## Prototype

```
USBH_STATUS USBH_MTP_Reset(USBH_MTP_DEVICE_HANDLE hDevice);
```

## Parameters

| Parameter | Description |
| --- | --- |
| `hDevice` | Handle to the opened device. |

## Return value

| | |
| --- | --- |
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 9.2.0.9 USBH_MTP_SetTimeouts()

### Description

Sets timeouts for read and write transactions for a device. The timeouts are valid for single transactions, not for whole API calls.

### Prototype

```
USBH_STATUS USBH_MTP_SetTimeouts(USBH_MTP_DEVICE_HANDLE hDevice,
                                 U32                    ReadTimeout,
                                 U32                    WriteTimeout);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ReadTimeout | Timeout for all transactions which read from the device. |
| WriteTimeout | Timeout for all transactions which write to the device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### Additional information

It is advised to set the timeouts to at least 10 seconds, as this is the time many Android devices may require to respond to certain commands.

## 9.2.0.10 USBH_MTP_GetLastErrorCode()

### Description

Returns the error code for the last executed operation.

### Prototype

```
U16 USBH_MTP_GetLastErrorCode(USBH_MTP_DEVICE_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

= 0 Last operation completed without an error code.
≠ 0 Error code. See USBH_MTP_RESPONSE_CODES for a list of MTP error codes.

## 9.2.0.11 USBH_MTP_GetStorageInfo()

### Description

Retrieves information about a storage on the device.

### Prototype

```
USBH_STATUS USBH_MTP_GetStorageInfo(USBH_MTP_DEVICE_HANDLE   hDevice,
                                    U8                       StorageIndex,
                                    USBH_MTP_STORAGE_INFO  * pStorageInfo);
```

### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to the opened device. |
| StorageIndex | Zero-based index of the storage, see USBH_MTP_GetNumStorages(). |
| pStorageInfo | out Pointer to a USBH_MTP_STORAGE_INFO structure to store information related to the storage. |

### Return value

| | |
| --- | --- |
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### Notes

This operation is always supported by MTP devices.

### 9.2.0.12 USBH_MTP_Format()

#### Description

Formats (deletes all data!) on a device storage.

#### Prototype

```
USBH_STATUS USBH_MTP_Format(USBH_MTP_DEVICE_HANDLE hDevice,
                            U8                     StorageIndex);
```

#### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| StorageIndex | Zero-based index of the storage, see USBH_MTP_GetNumStorages(). |

#### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 9.2.0.13 USBH_MTP_GetNumObjects()

### Description

Retrieves the number of objects inside a single directory.

### Prototype

```
USBH_STATUS USBH_MTP_GetNumObjects(USBH_MTP_DEVICE_HANDLE   hDevice,
                                   U8                       StorageIndex,
                                   U32                      DirObjectID,
                                   U32                    * pNumObjects);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| StorageIndex | Zero-based index of the storage, see USBH_MTP_GetNumStorages(). |
| DirObjectID | Object ID for the directory. |
| pNumObjects | out Pointer to a variable where the number of objects inside the directory will be stored. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 9.2.0.14 USBH_MTP_GetObjectList()

### Description

Retrieves a list of object IDs from a directory. The number of objects inside a directory can be found out beforehand by using `USBH_MTP_GetNumObjects`.

### Prototype

```
USBH_STATUS USBH_MTP_GetObjectList(USBH_MTP_DEVICE_HANDLE   hDevice,
                                   U8                       StorageIndex,
                                   U32                      DirObjectID,
                                   USBH_MTP_OBJECT        * pBuffer,
                                   U32                    * pNumObjects);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| StorageIndex | Zero-based index of the storage, see `USBH_MTP_GetNumStorages()`. |
| DirObjectID | Object ID for the directory. |
| pBuffer | out Pointer to an array of `USBH_MTP_OBJECT` structures. |
| pNumObjects | in/out The application should specify the size of the buffer in `USBH_MTP_OBJECT` units. The MTP module will read object IDs up to the specified value. If there are less objects in the folder the number of objects read will be stored in this variable. If there are more objects in the folder than specified the data for the surplus objects is discarded by the module. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Example

```
static USBH_MTP_OBJECT _aObjBuffer[10];
<...>
Status = USBH_MTP_GetNumObjects(hDevice, StorageIndex, DirObjectID, &NumObjectsDir);
if (Status == USBH_STATUS_SUCCESS) {
  USBH_Logf_Application("Found %d objects in directory 0x%0.8X \n",
                        NumObjectsDir, DirObjectID);
  NumObjects = USBH_MIN(NumObjectsDir, NumObjectsFree);
  //
  // Retrieve a list of object IDs from the root directory.
  //
  Status = USBH_MTP_GetObjectList(hDevice,
                                  StorageIndex,
                                  DirObjectID,
                                  _aObjBuffer,
                                  &NumObjects);
  if (Status == USBH_STATUS_SUCCESS) {
    <...>
  } else {
    <...>
  }
} else {
  <...>
}
```

## 9.2.0.15 USBH_MTP_GetObjectInfo()

### Description

Retrieves the ObjectInfo dataset for a specific object.

### Prototype

```
USBH_STATUS USBH_MTP_GetObjectInfo(USBH_MTP_DEVICE_HANDLE   hDevice,
                                   U32                      ObjectID,
                                   USBH_MTP_OBJECT_INFO   * pObjInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectID | Object ID to retrieve information for. |
| pObjInfo | out Pointer to a USBH_MTP_OBJECT_INFO structure where the data will be stored. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 9.2.0.16 USBH_MTP_CreateObject()

### Description

Writes a new object onto the device. MTP does not allow files to be written in chunks, therefore a callback mechanism is implemented to allow the embedded host to write files of any size onto the MTP device. As soon as the contents of the first buffer have been written or the file has been completely written onto the device - the registered callback is called. Inside the callback the user can either put new data into the previously used buffer or change the buffer by modifying the pNextBuffer parameter inside the USBH_SEND_DATA_FUNC callback. Using two (or more) buffers and switching between them has the advantage that the MTP module can write continuously to the device.

### Prototype

```
USBH_STATUS USBH_MTP_CreateObject(USBH_MTP_DEVICE_HANDLE   hDevice,
                                  U8                       StorageIndex,
                                  USBH_MTP_CREATE_INFO   * pInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| StorageIndex | Zero-based index of the storage, see USBH_MTP_GetNumStorages(). |
| pInfo | in/out Pointer to a USBH_MTP_CREATE_INFO structure where parameters for the new object are stored. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. On success the member ObjectID of the USBH_MTP_CREATE_INFO will contain the new object ID provided by the device. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### Example

```
static U8 _acBufWrite[1024*64];
const U16 _sFileName[] = L"SEGGER.txt";
U32 FileSize = 1024 * 1024;

/*********************************************************************
*
*       _SendData
*
*  Function description
*    In this sample application the file data is simply generated
*    through a memset, in a real application data can for example
*    be read from the host's file system.
*/
static void _SendData(void * pUserContext,
                      U32 NumBytesSentTotal,
                      U32 * pNumBytesToSend,
                      void ** pNextBuffer) {
  U32 NumBytesToSend;
  int r;

  NumBytesToSend = *(U32*)&pUserContext - NumBytesSentTotal;
  NumBytesToSend = USBH_MIN(sizeof(_acBufWrite), NumBytesToSend);
  if (NumBytesToSend) {
    USBH_MEMSET(_acBufWrite, 0xA5, NumBytesToSend);
  }
}
```

```
<...>
USBH_MTP_CREATE_INFO CreateInfo;

CreateInfo.FileNameSize   = strlen(_sFileName) + 1; // File name length
                                                    // + terminating character
CreateInfo.sFileName      = _sFileName;             // File name Unicode string
CreateInfo.ObjectSize     = FileSize;               // Size of the file in bytes
CreateInfo.ParentObjectID = ObjectID;               // Object ID of the parent
                                                    // folder
CreateInfo.pfGetData      = _SendData;              // Callback function
CreateInfo.pUserBuf       = _acBufWrite;            // User buffer containing
                                                    // the data
CreateInfo.UserBufSize    = sizeof(_acBufWrite);    // Size of the user buffer
CreateInfo.isFolder       = FALSE;                  // Not a folder
CreateInfo.pUserContext   = (void*)FileSize;        // User context is passed
                                                    // to the callback

Status = USBH_MTP_CreateObject(hDevice, StorageIndex, &CreateInfo);
<...>
```

## 9.2.0.17 USBH_MTP_DeleteObject()

### Description

Deletes an object from the device.

### Prototype

```
USBH_STATUS USBH_MTP_DeleteObject(USBH_MTP_DEVICE_HANDLE hDevice,
                                  U32                    ObjectID);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectID | Object ID to be deleted. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 9.2.0.18 USBH_MTP_Rename()

### Description

Changes the name of an object.

### Prototype

```
USBH_STATUS USBH_MTP_Rename(      USBH_MTP_DEVICE_HANDLE   hDevice,
                                  U32                      ObjectID,
                            const U16                    * sNewName,
                                  U32                      NumChars);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectID | Object ID to retrieve the property from. |
| sNewName | Pointer to a Unicode string containing the new file name. |
| NumChars | Length of the new file name in U16 units. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 9.2.0.19 USBH_MTP_ReadFile()

### Description

Reads a file from the device. MTP does not allow files to be read in chunks, therefore a callback mechanism is implemented to allow embedded devices with limited memory to be able to read files of any size from an MTP device. The callback is called as soon as the user provided buffer is full or the file has been completely read. In the callback the user can either process the data in the user buffer or change the user buffer by writing the `pNextBuffer` parameter inside the `USBH_RECEIVE_DATA_FUNC` callback and process the data in the first buffer in another task. The second method has the advantage that the callback can return immediately and the MTP module can continue reading from the device.

### Prototype

```
USBH_STATUS USBH_MTP_ReadFile(USBH_MTP_DEVICE_HANDLE   hDevice,
                              U32                      ObjectID,
                              USBH_RECEIVE_DATA_FUNC * pfReadData,
                              void                   * pUserContext,
                              U8                     * pUserBuf,
                              U32                      UserBufSize);
```

### Parameters

| Parameter | Description |
|---|---|
| `hDevice` | Handle to the opened device. |
| `ObjectID` | Object ID to retrieve the property from. |
| `pfReadData` | Pointer to a user-provided `USBH_RECEIVE_DATA_FUNC` callback function which will be called when the file is being received. |
| `pUserContext` | Pointer to a user context which is passed to the callback function. |
| `pUserBuf` | Pointer to a buffer where the data will be stored. This parameter can be `NULL`. In this case the callback is called directly and a buffer has to be set from there. |
| `UserBufSize` | Size of the user buffer. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### Example

```
/*********************************************************************
*
*       _ReceiveData
*
*  Function description
*    This function is responsible for receiving the data provided by
*    the USBH_MTP_ReadFile function.
*/
static void _ReceiveData(void * pUserContext,
                         U32 NumBytesRemaining,
                         U32 NumBytesInBuffer,
                         void ** pNextBuffer,
                         U32 * pNextBufferSize) {
  printf("Reading file 0x%8.8lX, reading chunk of %lu still have to "
         "read %lu bytes for the current file.\n",
         *(U32*)pUserContext,
         NumBytesInBuffer,
         NumBytesRemaining);
```

```
}

<...>

Status = USBH_MTP_ReadFile(hDevice, // Handle to the device
                           ObjectID, // Object ID to read
                           &_ReceiveData, // Callback function
                           &ObjectID, // User context passed to the callback
                           _acReadBuf, // User buffer to use
                           sizeof(_acReadBuf)); // User buffer size
```

## 9.2.0.20 USBH_MTP_GetDevicePropDesc()

### Description

Retrieves the description of a MTP property from the device. The description includes the size of a property, which is highly important as the same properties can have different sizes on different devices.

### Prototype

```
USBH_STATUS USBH_MTP_GetDevicePropDesc(USBH_MTP_DEVICE_HANDLE      hDevice,
                                       U16                         DevicePropCode,
                                       USBH_MTP_DEVICE_PROP_DESC * pDesc);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| DevicePropCode | Device property code, see USBH_MTP_DEVICE_PROPERTIES. |
| pDesc | Pointer to a USBH_MTP_DEVICE_PROP_DESC structure where the information should be saved. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 9.2.0.21 USBH_MTP_GetDevicePropValue()

### Description

Retrieves the value of a property of a specific Device. The property description has to be retrieved via USBH_MTP_GetDevicePropDesc prior to calling this function.

### Prototype

```
USBH_STATUS USBH_MTP_GetDevicePropValue
                                    (       USBH_MTP_DEVICE_HANDLE      hDevice,
                                     const USBH_MTP_DEVICE_PROP_DESC * pDesc,
                                           void                      * pData,
                                           U32                         BufferSize);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pDesc | Pointer to a USBH_MTP_DEVICE_PROP_DESC structure which has the property size and code. |
| pData | Pointer to a buffer where the value should be stored. |
| BufferSize | Size of the buffer, if the value is longer than the size of the buffer the value will be truncated. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 9.2.0.22 USBH_MTP_GetObjectPropsSupported()

## Description

Retrieves a list of supported properties for a given object format.

## Prototype

```
USBH_STATUS USBH_MTP_GetObjectPropsSupported
                                       (USBH_MTP_DEVICE_HANDLE   hDevice,
                                        U32                      ObjectFormatCode,
                                        U16                    * pBuffer,
                                        U32                    * pNumProps);
```

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectFormatCode | MTP Format Code ID for the MTP property which should be queried. (See USBH_MTP_OBJECT_FORMAT for a list of valid format codes). |
| pBuffer | out Pointer to an array of U16 values, this array will receive the list of property codes. |
| pNumProps | in/out The application should specify the size of the buffer in U16 values. The MTP module will read property codes up to the specified value. If there are less codes delivered by the device the number of codes read will be stored in this variable. If there are more codes delivered by device the surplus codes are discarded by the module. |

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## Additional information

Unfortunately there is no way to ask the device how many properties a format has before requesting the list or to request a partial list, therefore the buffer should be big enough to contain all of them.

## 9.2.0.23 USBH_MTP_GetObjectPropDesc()

### Description

Retrieves information about an MTP object property used by the device. This is especially important because the application needs to know the data type (the size) of the property before retrieving it.

### Prototype

```
USBH_STATUS USBH_MTP_GetObjectPropDesc
                                    (USBH_MTP_DEVICE_HANDLE      hDevice,
                                     U16                         ObjectPropCode,
                                     U32                         ObjectFormatCode,
                                     USBH_MTP_OBJECT_PROP_DESC * pDesc);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectPropCode | Object property code, see USBH_MTP_OBJECT_PROPERTIES. |
| ObjectFormatCode | MTP Format Code ID for the MTP property which should be queried. (See USBH_MTP_OBJECT_FORMAT for a list of valid format codes). |
| pDesc | in Pointer to a USBH_MTP_OBJECT_PROP_DESC structure where the MTP property descriptor will be stored. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### Example

```
/*********************************************************************
*
* _DataTypeToBytes
*
* Function description
* Returns the number of bytes required for the given data type.
*/
static unsigned _DataTypeToBytes(U16 DataType) {
  unsigned NumBytes;

  switch (DataType) {
  case USBH_MTP_DATA_TYPE_INT8:
  case USBH_MTP_DATA_TYPE_UINT8:
    NumBytes = 1;
    break;
  case USBH_MTP_DATA_TYPE_INT16:
  case USBH_MTP_DATA_TYPE_UINT16:
    NumBytes = 2;
    break;
  case USBH_MTP_DATA_TYPE_INT32:
  case USBH_MTP_DATA_TYPE_UINT32:
    NumBytes = 4;
    break;
  case USBH_MTP_DATA_TYPE_INT64:
  case USBH_MTP_DATA_TYPE_UINT64:
    NumBytes = 8;
    break;
  case USBH_MTP_DATA_TYPE_STR:
    NumBytes = 256;
```

```
    break;
  case USBH_MTP_DATA_TYPE_UINT128:
    NumBytes = 16;
    break;
  default:
    NumBytes = 0;// Error, invalid data type.
    break;
  }
  return NumBytes;
}

USBH_MTP_OBJECT_PROP_DESC ObjPropDesc;
U8 * pPropertyBuffer;

//
// Check in which format the property
// USBH_MTP_OBJECT_PROP_STORAGE_ID is stored.
//
Status = USBH_MTP_GetObjectPropDesc(hDevice,
                                   USBH_MTP_OBJECT_PROP_STORAGE_ID,
                                   USBH_MTP_OBJECT_FORMAT_UNDEFINED,
                                   &ObjPropDesc);
if (Status == USBH_STATUS_SUCCESS) {
  //
  // Now that we know the format - memory can be allocated
  // and the property can be retrieved.
  //
  NumBytes = _DataTypeToBytes(ObjPropDesc.DataType);
  pPropertyBuffer = malloc(NumBytes);
  if (pPropertyBuffer) {
    Status = USBH_MTP_GetObjectPropValue(hDevice,
                                        CreateInfo.ObjectID,
                                        &ObjPropDesc,
                                        pPropertyBuffer,
                                        NumBytes);
    if (Status == USBH_STATUS_SUCCESS) {
      <...do something with the value...>
    } else {
      <...>
    }
    free(pPropertyBuffer);
  } else {
    <...>
  }
} else {
  <...>
}
```

## 9.2.0.24 USBH_MTP_GetObjectPropValue()

### Description

Retrieves the value of a property of a specific object. The property description has to be retrieved via USBH_MTP_GetObjectPropDesc prior to calling this function.

### Prototype

```
USBH_STATUS USBH_MTP_GetObjectPropValue
                                        (       USBH_MTP_DEVICE_HANDLE      hDevice,
                                                U32                         ObjectID,
                                          const USBH_MTP_OBJECT_PROP_DESC * pDesc,
                                                void                      * pData,
                                                U32                         BufferSize);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectID | Object ID to retrieve the property from. |
| pDesc | Pointer to a USBH_MTP_OBJECT_PROP_DESC structure which has the property size and code. |
| pData | Pointer to a buffer where the value should be stored. |
| BufferSize | Size of the buffer, if the value is longer than the size of the buffer the value will be truncated. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Example

See USBH_MTP_GetObjectPropDesc().

## 9.2.0.25 USBH_MTP_SetObjectProperty()

### Description

Sets the property of an object to the specified value.

### Prototype

```
USBH_STATUS USBH_MTP_SetObjectProperty(       USBH_MTP_DEVICE_HANDLE      hDevice,
                                              U32                         ObjectID,
                                        const USBH_MTP_OBJECT_PROP_DESC * pDesc,
                                        const void                      * pData,
                                              U32                         NumBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ObjectID | Object ID to retrieve the property from. |
| pDesc | Pointer to a USBH_MTP_OBJECT_PROP_DESC structure which should be retrieved earlier via USBH_MTP_GetObjectPropDesc(). |
| pData | Pointer to a buffer where the new value is stored. |
| NumBytes | Size of the value inside the buffer in bytes. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Example

```
USBH_MTP_OBJECT_PROP_DESC ObjPropDesc;
const U16 _sNewDateCreated[] = L"20010101T011642.0-0700";

//
// Change "DateCreated"
//
ObjPropDesc.PropertyCode = USBH_MTP_OBJECT_PROP_DATE_CREATED;
ObjPropDesc.DataType = USBH_MTP_DATA_TYPE_STR;
Status = USBH_MTP_SetObjectProperty(hDevice,
                                    CreateInfo.ObjectID,
                                    &ObjPropDesc,
                                    (void *)_sNewDateCreated,
                                    sizeof(_sNewDateCreated));
```

## 9.2.0.26 USBH_MTP_CheckLock()

### Description

Determines whether the device is locked by a pin/password/etc.

### Prototype

```
USBH_STATUS USBH_MTP_CheckLock(USBH_MTP_DEVICE_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | The device is not locked. |
| USBH_STATUS_ERROR | The device is locked. |
| USBH_STATUS_BUSY | The endpoint is busy with a different operation. |

Any other value means the device is locked and reports the specific error through which it was determined.

### Additional information

Some devices (mainly Windows Phone) may re-enumerate when they are unlocked by the user, which will cause this function to correctly report `USBH_STATUS_DEVICE_REMOVED`. The application should open a new handle to the device and call this function again.

On some devices (mainly Android) the storage count of the device (the value which you get from `USBH_MTP_GetNumStorages()`) will be updated automatically when this function is called and the user has unlocked the device (normally from zero to the real value).

## 9.2.0.27 USBH_MTP_SetEventCallback()

### Description

Sets a callback for MTP events, e.g. StoreAdded, ObjectAdded, etc.

### Prototype

```
USBH_STATUS USBH_MTP_SetEventCallback(USBH_MTP_DEVICE_HANDLE   hDevice,
                                      USBH_EVENT_CALLBACK    * cbOnUserEvent);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| cbOnUserEvent | Pointer to a user provided function of type USBH_EVENT_CALLBACK. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Additional information

The callback should not block. See the description of USBH_EVENT_CALLBACK for additional information.

## 9.2.0.28 USBH_MTP_ConfigEventSupport()

### Description

Turns MTP event support on or off. Should be called after `USBH_MTP_Init()` When turning MTP support on or off only newly connected devices will be affected. Event support is off by default.

### Prototype

```
void USBH_MTP_ConfigEventSupport(U8 OnOff);
```

### Parameters

| Parameter | Description |
|---|---|
| OnOff | Turn events on or off. |

### Additional information

Calling this function will not affect devices which are already open. To make sure open devices are affected you have to close them (`USBH_MTP_Close()`) and open them again (`USBH_MTP_Open()`). Some MTP devices do not behave as they should when re-opening a session and refuse to communicate, in such a case it is advised to re-enumerate them.

## 9.2.0.29 USBH_MTP_GetEventSupport()

### Description

Returns the event support configuration, see `USBH_MTP_ConfigEventSupport()` for details.

### Prototype

```
U8 USBH_MTP_GetEventSupport(void);
```

### Return value

| | |
|---|---|
| 1 | Event support enabled. |
| 0 | Event support disabled. |

## 9.3 Data structures

This chapter describes the emUSB-Host HID API structures.

| Structure | Description |
|---|---|
| USBH_MTP_DEVICE_INFO | Contains information about an MTP compatible device. |
| USBH_MTP_STORAGE_INFO | Contains information about an MTP storage. |
| USBH_MTP_OBJECT | Contains basic information about an MTP object. |
| USBH_MTP_OBJECT_INFO | Contains extended information about an MTP object. |
| USBH_MTP_CREATE_INFO | Contains information needed to create a new MTP object. |
| USBH_MTP_OBJECT_PROP_DESC | Contains information about the data-type and accessibility of an object property. |

# 9.3.0.1 USBH_MTP_DEVICE_INFO

## Description

Contains information about an MTP compatible device.

## Type definition

```
typedef struct {
  U16          VendorId;
  U16          ProductId;
  U8           acSerialNo[];
  USBH_SPEED   Speed;
  const U16 *  sManufacturer;
  const U16 *  sModel;
  const U16 *  sDeviceVersion;
  const U16 *  sSerialNumber;
} USBH_MTP_DEVICE_INFO;
```

## Structure members

| Member | Description |
|---|---|
| VendorId | Vendor identification number. |
| ProductId | Product identification number. |
| acSerialNo | Serial number string. |
| Speed | The USB speed of the device, see USBH_SPEED. |
| sManufacturer | Pointer to a Unicode string “manufacturer”. |
| sModel | Pointer to a Unicode string “model”. |
| sDeviceVersion | Pointer to a Unicode string “device version”. |
| sSerialNumber | Pointer to a Unicode string “serial number”. |

### 9.3.0.2 USBH_MTP_STORAGE_INFO

#### Description

Contains information about an MTP storage.

#### Type definition

```
typedef struct {
  U16  StorageType;
  U16  FilesystemType;
  U16  AccessCapability;
  U8   MaxCapacity[];
  U8   FreeSpaceInBytes[];
  U32  FreeSpaceInImages;
  U16  sStorageDescription[];
  U16  sVolumeLabel[];
} USBH_MTP_STORAGE_INFO;
```

#### Structure members

| Member | Description |
|---|---|
| StorageType | 0x0000 Undefined<br>0x0001 Fixed ROM<br>0x0002 Removable ROM<br>0x0003 Fixed RAM<br>0x0004 Removable RAM<br>Note: This value is often unreliable as many devices return 0x0003 for everything. |
| FilesystemType | 0x0000 Undefined<br>0x0001 Generic flat<br>0x0002 Generic hierarchical<br>0x0003 DCF |
| AccessCapability | 0x0000 Read-write<br>0x0001 Read-only without object deletion<br>0x0002 Read-only with object deletion |
| MaxCapacity | An U64 little-endian value which designates the maximum capacity of the storage in bytes. The value is declared as an array of U8, you will have to cast it into a U64 in your application. If the storage is read-only, this field is optional and may contain zero. |
| FreeSpaceInBytes | An U64 little-endian value which designates the free space on the storage in bytes. The value is declared as an array of U8, you will have to cast it into a U64 in your application. If the storage is read-only, this field is optional and may contain zero. |
| FreeSpaceInImages | Value describing the number of images which could still be fit into the storage. This is a PTP relevant value, for MTP it is normally zero or 0xFFFFFFFF. |
| sStorageDescription | Unicode string describing the storage. |
| sVolumeLabel | Unicode string which contains the volume label. |

### 9.3.0.3 USBH_MTP_OBJECT

#### Description

Contains basic information about an MTP object.

#### Type definition

```
typedef struct {
  U32  ObjectID;
  U16  ObjectFormat;
  U16  AssociationType;
} USBH_MTP_OBJECT;
```

#### Structure members

| Member | Description |
|---|---|
| ObjectID | Unique ID for the object, provided by the device. |
| ObjectFormat | MTP Object format, see USBH_MTP_OBJECT_FORMAT |
| AssociationType | MTP association type, see USBH_MTP_ASSOCIATION_TYPES |

### 9.3.0.4 USBH_MTP_OBJECT_INFO

#### Description

Contains extended information about an MTP object.

#### Type definition

```
typedef struct {
  U32  StorageID;
  U16  ObjectFormat;
  U16  ProtectionStatus;
  U32  ParentObject;
  U16  AssociationType;
  U16  sFilename[];
  U16  sCaptureDate[];
  U16  sModificationDate[];
} USBH_MTP_OBJECT_INFO;
```

#### Structure members

| Member | Description |
|---|---|
| StorageID | ID of the storage where the Object is located. |
| ObjectFormat | MTP Object format, see USBH_MTP_OBJECT_FORMAT. |
| ProtectionStatus | 0x0000 - No protection<br>0x0001 - Read-only<br>0x8002 - Read-only data<br>0x8003 - Non-transferable data |
| ParentObject | ObjectID of the parent Object. For "root" use the define USBH_MTP_ROOT_OBJECT_ID. |
| AssociationType | MTP association type, see USBH_MTP_ASSOCIATION_TYPES |
| sFilename | File name buffer. |
| sCaptureDate | CaptureDate string buffer. |
| sModificationDate | ModificationDate string buffer. |

## 9.3.0.5 USBH_MTP_CREATE_INFO

### Description

Contains information needed to create a new MTP object.

### Type definition

```
typedef struct {
  U32                    ObjectID;
  U32                    ParentObjectID;
  U32                    ObjectSize;
  USBH_SEND_DATA_FUNC  * pfGetData;
  U32                    FileNameSize;
  const U16            * sFileName;
  U8                     isFolder;
  U8                   * pUserBuf;
  U32                    UserBufSize;
  void                 * pUserContext;
} USBH_MTP_CREATE_INFO;
```

### Structure members

| Member | Description |
|---|---|
| ObjectID | Filled by the MTP Module after the Object has been created. This ID is provided by the device. |
| ParentObjectID | The ObjectID of the parent object. For "root" use USBH_MTP_ROOT_OBJECT_ID. |
| ObjectSize | Size of the file in bytes. |
| pfGetData | Pointer to a user-provided callback which will provide the data. See USBH_SEND_DATA_FUNC. |
| FileNameSize | The length of the file-name string. |
| sFileName | Pointer to the Unicode file-name string. |
| isFolder | Flag indicating whether the new object should be a folder or not. When creating a folder pfGetData, pUserBuf, UserBufSize can be set to NULL. |
| pUserBuf | Pointer to a user buffer where the data is located. Can be NULL, in this case the callback is called immediately and the buffer has to be set inside the callback. |
| UserBufSize | Size of the user buffer in bytes. |
| pUserContext | User context which is passed to the callback. |

### 9.3.0.6 USBH_MTP_OBJECT_PROP_DESC

#### Description

Contains information about the data-type and accessibility of an object property.

#### Type definition

```
typedef struct {
  U16  PropertyCode;
  U16  DataType;
  U8   GetSet;
} USBH_MTP_OBJECT_PROP_DESC;
```

#### Structure members

| Member | Description |
|---|---|
| PropertyCode | MTP object property code, see USBH_MTP_OBJECT_PROP-ERTIES. |
| DataType | Data type of the property. |
| GetSet | 0 - Read-only, 1 - Read-write. |

#### Additional information

| Type | Code | Size in bytes |
|---|---|---|
| USBH_MTP_DATA_TYPE_INT8 | 0x0001 | 1 |
| USBH_MTP_DATA_TYPE_UINT8 | 0x0002 | 1 |
| USBH_MTP_DATA_TYPE_INT16 | 0x0003 | 2 |
| USBH_MTP_DATA_TYPE_UINT16 | 0x0004 | 2 |
| USBH_MTP_DATA_TYPE_INT32 | 0x0005 | 4 |
| USBH_MTP_DATA_TYPE_UINT32 | 0x0006 | 4 |
| USBH_MTP_DATA_TYPE_INT64 | 0x0007 | 8 |
| USBH_MTP_DATA_TYPE_UINT64 | 0x0008 | 8 |
| USBH_MTP_DATA_TYPE_UINT128 | 0x000A | 16 |
| USBH_MTP_DATA_TYPE_AUINT8 | 0x4002 | Variable size. |
| USBH_MTP_DATA_TYPE_STR | 0xFFFF | Variable size. |

# 9.4 Function Types

This chapter describes the emUSB-Host MTP API function types.

| Type | Description |
|---|---|
| USBH_SEND_DATA_FUNC | Definition of the callback which has to be specified when using USBH_MTP_CreateObject(). |
| USBH_RECEIVE_DATA_FUNC | Definition of the callback which has to be specified when using USBH_MTP_ReadFile(). |
| USBH_EVENT_CALLBACK | Definition of the callback which can be set via USBH_MTP_SetEventCallback(). |

## 9.4.0.1 USBH_SEND_DATA_FUNC

### Description

Definition of the callback which has to be specified when using `USBH_MTP_CreateObject()`.

### Type definition

```
typedef void USBH_SEND_DATA_FUNC(void * pUserContext,
                                  U32      NumBytesSentTotal,
                                  U32    * pNumBytesToSend,
                                  U8   * * ppNextBuffer);
```

### Parameters

| Parameter | Description |
|---|---|
| pUserContext | User context which is passed to the callback. |
| NumBytesSentTotal | This value contains the total number of bytes which have already been transferred |
| pNumBytesToSend | The user has to set this value to the number of bytes which are inside the buffer. |
| ppNextBuffer | The user can change this pointer to a different buffer. If this parameter remains NULL after the callback returns, the previous buffer is re-used (the application should put new data into the buffer first). |

## 9.4.0.2 USBH_RECEIVE_DATA_FUNC

### Description

Definition of the callback which has to be specified when using `USBH_MTP_ReadFile()`.

### Type definition

```
typedef void USBH_RECEIVE_DATA_FUNC(void  * pUserContext,
                                    U32     NumBytesRemaining,
                                    U32     NumBytesInBuffer,
                                    void ** ppNextBuffer,
                                    U32   * pNextBufferSize);
```

### Parameters

| Parameter | Description |
|---|---|
| `pUserContext` | User context which is passed to the callback. |
| `NumBytesRemaining` | This value contains the total number of bytes which still have to be read. |
| `NumBytesInBuffer` | The number of bytes which have been read in this transaction. |
| `ppNextBuffer` | The user can change this pointer to a different buffer. If this parameter remains `NULL` after the callback returns, the previous buffer is re-used (the application should copy the data out of the buffer first, as it will be overwritten on the next transaction). |
| `pNextBufferSize` | Size of the next buffer. This only needs to be changed when the `pNextBuffer` parameter is changed. |

### 9.4.0.3 USBH_EVENT_CALLBACK

#### Description

Definition of the callback which can be set via USBH_MTP_SetEventCallback().

#### Type definition

```
typedef void USBH_EVENT_CALLBACK(U16 EventCode,
                                 U32 Para1,
                                 U32 Para2,
                                 U32 Para3);
```

#### Parameters

| Parameter | Description |
|---|---|
| EventCode | Code of the MTP event, see USBH_MTP_EVENT_CODES. |
| Para1 | First parameter passed with the event. |
| Para2 | Second parameter passed with the event. |
| Para3 | Third parameter passed with the event. |

#### Additional information

The events USBH_MTP_EVENT_STORE_ADDED and USBH_MTP_EVENT_STORE_REMOVED are handled by the MTP module before being passed to the callback. The storage information for the device is updated automatically when one of these events is received. All events are passed to the callback, this includes vendor specific events which are not present in the USBH_MTP_EVENT_CODES enum. Parameters which are not used with a specific event (e.g. USBH_MTP_EVENT_STORE_ADDED has only one parameter) will be passed as zero. The callback should not block.

# 9.5 Enums

This chapter describes the emUSB-Host MTP API enums.

| Enum | Description |
|---|---|
| USBH_MTP_DEVICE_PROPERTIES | Device properties describe conditions or setting relevant to the device itself. |
| USBH_MTP_OBJECT_PROPERTIES | Object properties identify settings or state conditions of files and folders (objects). |
| USBH_MTP_RESPONSE_CODES | Possible response codes reported by the device upon completion of an operation. |
| USBH_MTP_OBJECT_FORMAT | Identifiers describing the format type of a given object. |
| USBH_MTP_EVENT_CODES | Events are described by a 16-bit code. |

### 9.5.0.1 USBH_MTP_DEVICE_PROPERTIES

#### Description

Device properties describe conditions or setting relevant to the device itself. The properties are unrelated to objects.

#### Type definition

```
typedef enum {
  USBH_MTP_DEVICE_PROP_UNDEFINED,
  USBH_MTP_DEVICE_PROP_BATTERY_LEVEL,
  USBH_MTP_DEVICE_PROP_FUNCTIONAL_MODE,
  USBH_MTP_DEVICE_PROP_IMAGE_SIZE,
  USBH_MTP_DEVICE_PROP_COMPRESSION_SETTING,
  USBH_MTP_DEVICE_PROP_WHITE_BALANCE,
  USBH_MTP_DEVICE_PROP_RGB_GAIN,
  USBH_MTP_DEVICE_PROP_F_NUMBER,
  USBH_MTP_DEVICE_PROP_FOCAL_LENGTH,
  USBH_MTP_DEVICE_PROP_FOCUS_DISTANCE,
  USBH_MTP_DEVICE_PROP_FOCUS_MODE,
  USBH_MTP_DEVICE_PROP_EXPOSURE_METERING_MODE,
  USBH_MTP_DEVICE_PROP_FLASH_MODE,
  USBH_MTP_DEVICE_PROP_EXPOSURE_TIME,
  USBH_MTP_DEVICE_PROP_EXPOSURE_PROGRAM_MODE,
  USBH_MTP_DEVICE_PROP_EXPOSURE_INDEX,
  USBH_MTP_DEVICE_PROP_EXPOSURE_BIAS_COMPENSATION,
  USBH_MTP_DEVICE_PROP_DATETIME,
  USBH_MTP_DEVICE_PROP_CAPTURE_DELAY,
  USBH_MTP_DEVICE_PROP_STILL_CAPTURE_MODE,
  USBH_MTP_DEVICE_PROP_CONTRAST,
  USBH_MTP_DEVICE_PROP_SHARPNESS,
  USBH_MTP_DEVICE_PROP_DIGITAL_ZOOM,
  USBH_MTP_DEVICE_PROP_EFFECT_MODE,
  USBH_MTP_DEVICE_PROP_BURST_NUMBER,
  USBH_MTP_DEVICE_PROP_BURST_INTERVAL,
  USBH_MTP_DEVICE_PROP_TIMELAPSE_NUMBER,
  USBH_MTP_DEVICE_PROP_TIMELAPSE_INTERVAL,
  USBH_MTP_DEVICE_PROP_FOCUS_METERING_MODE,
  USBH_MTP_DEVICE_PROP_UPLOAD_URL,
  USBH_MTP_DEVICE_PROP_ARTIST,
  USBH_MTP_DEVICE_PROP_COPYRIGHT_INFO,
  USBH_MTP_DEVICE_PROP_SYNCHRONIZATION_PARTNER,
  USBH_MTP_DEVICE_PROP_DEVICE_FRIENDLY_NAME,
  USBH_MTP_DEVICE_PROP_VOLUME,
  USBH_MTP_DEVICE_PROP_SUPPORTEDFORMATSORDERED,
  USBH_MTP_DEVICE_PROP_DEVICEICON,
  USBH_MTP_DEVICE_PROP_PLAYBACK_RATE,
  USBH_MTP_DEVICE_PROP_PLAYBACK_OBJECT,
  USBH_MTP_DEVICE_PROP_PLAYBACK_CONTAINER,
  USBH_MTP_DEVICE_PROP_SESSION_INITIATOR_VERSION_INFO,
  USBH_MTP_DEVICE_PROP_PERCEIVED_DEVICE_TYPE
} USBH_MTP_DEVICE_PROPERTIES;
```

## 9.5.0.2 USBH_MTP_OBJECT_PROPERTIES

### Description

Object properties identify settings or state conditions of files and folders (objects).

### Type definition

```
typedef enum {
  USBH_MTP_OBJECT_PROP_STORAGE_ID,
  USBH_MTP_OBJECT_PROP_OBJECT_FORMAT,
  USBH_MTP_OBJECT_PROP_PROTECTION_STATUS,
  USBH_MTP_OBJECT_PROP_OBJECT_SIZE,
  USBH_MTP_OBJECT_PROP_ASSOCIATION_TYPE,
  USBH_MTP_OBJECT_PROP_ASSOCIATION_DESC,
  USBH_MTP_OBJECT_PROP_OBJECT_FILE_NAME,
  USBH_MTP_OBJECT_PROP_DATE_CREATED,
  USBH_MTP_OBJECT_PROP_DATE_MODIFIED,
  USBH_MTP_OBJECT_PROP_KEYWORDS,
  USBH_MTP_OBJECT_PROP_PARENT_OBJECT,
  USBH_MTP_OBJECT_PROP_ALLOWED_FOLDER_CONTENTS,
  USBH_MTP_OBJECT_PROP_HIDDEN,
  USBH_MTP_OBJECT_PROP_SYSTEM_OBJECT,
  USBH_MTP_OBJECT_PROP_PERSISTENT_UNIQUE_OBJECT_IDENTIFIER,
  USBH_MTP_OBJECT_PROP_SYNCID,
  USBH_MTP_OBJECT_PROP_PROPERTY_BAG,
  USBH_MTP_OBJECT_PROP_NAME,
  USBH_MTP_OBJECT_PROP_CREATED_BY,
  USBH_MTP_OBJECT_PROP_ARTIST,
  USBH_MTP_OBJECT_PROP_DATE_AUTHORED,
  USBH_MTP_OBJECT_PROP_DESCRIPTION,
  USBH_MTP_OBJECT_PROP_URL_REFERENCE,
  USBH_MTP_OBJECT_PROP_LANGUAGELOCALE,
  USBH_MTP_OBJECT_PROP_COPYRIGHT_INFORMATION,
  USBH_MTP_OBJECT_PROP_SOURCE,
  USBH_MTP_OBJECT_PROP_ORIGIN_LOCATION,
  USBH_MTP_OBJECT_PROP_DATE_ADDED,
  USBH_MTP_OBJECT_PROP_NON_CONSUMABLE,
  USBH_MTP_OBJECT_PROP_CORRUPTUNPLAYABLE,
  USBH_MTP_OBJECT_PROP_PRODUCERSERIALNUMBER,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_FORMAT,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_SIZE,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_HEIGHT,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_WIDTH,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_DURATION,
  USBH_MTP_OBJECT_PROP_REPRESENTATIVE_SAMPLE_DATA,
  USBH_MTP_OBJECT_PROP_WIDTH,
  USBH_MTP_OBJECT_PROP_HEIGHT,
  USBH_MTP_OBJECT_PROP_DURATION,
  USBH_MTP_OBJECT_PROP_RATING,
  USBH_MTP_OBJECT_PROP_TRACK,
  USBH_MTP_OBJECT_PROP_GENRE,
  USBH_MTP_OBJECT_PROP_CREDITS,
  USBH_MTP_OBJECT_PROP_LYRICS,
  USBH_MTP_OBJECT_PROP_SUBSCRIPTION_CONTENT_ID,
  USBH_MTP_OBJECT_PROP_PRODUCED_BY,
  USBH_MTP_OBJECT_PROP_USE_COUNT,
  USBH_MTP_OBJECT_PROP_SKIP_COUNT,
  USBH_MTP_OBJECT_PROP_LAST_ACCESSED,
  USBH_MTP_OBJECT_PROP_PARENTAL_RATING,
  USBH_MTP_OBJECT_PROP_META_GENRE,
  USBH_MTP_OBJECT_PROP_COMPOSER,
  USBH_MTP_OBJECT_PROP_EFFECTIVE_RATING,
  USBH_MTP_OBJECT_PROP_SUBTITLE,
  USBH_MTP_OBJECT_PROP_ORIGINAL_RELEASE_DATE,
  USBH_MTP_OBJECT_PROP_ALBUM_NAME,
  USBH_MTP_OBJECT_PROP_ALBUM_ARTIST,
```

```
    USBH_MTP_OBJECT_PROP_MOOD,
    USBH_MTP_OBJECT_PROP_DRM_STATUS,
    USBH_MTP_OBJECT_PROP_SUB_DESCRIPTION,
    USBH_MTP_OBJECT_PROP_IS_CROPPED,
    USBH_MTP_OBJECT_PROP_IS_COLOUR_CORRECTED,
    USBH_MTP_OBJECT_PROP_IMAGE_BIT_DEPTH,
    USBH_MTP_OBJECT_PROP_FNUMBER,
    USBH_MTP_OBJECT_PROP_EXPOSURE_TIME,
    USBH_MTP_OBJECT_PROP_EXPOSURE_INDEX,
    USBH_MTP_OBJECT_PROP_TOTAL_BITRATE,
    USBH_MTP_OBJECT_PROP_BITRATE_TYPE,
    USBH_MTP_OBJECT_PROP_SAMPLE_RATE,
    USBH_MTP_OBJECT_PROP_NUMBER_OF_CHANNELS,
    USBH_MTP_OBJECT_PROP_AUDIO_BITDEPTH,
    USBH_MTP_OBJECT_PROP_SCAN_TYPE,
    USBH_MTP_OBJECT_PROP_AUDIO_WAVE_CODEC,
    USBH_MTP_OBJECT_PROP_AUDIO_BITRATE,
    USBH_MTP_OBJECT_PROP_VIDEO_FOURCC_CODEC,
    USBH_MTP_OBJECT_PROP_VIDEO_BITRATE,
    USBH_MTP_OBJECT_PROP_FRAMES_PER_THOUSAND_SECONDS,
    USBH_MTP_OBJECT_PROP_KEYFRAME_DISTANCE,
    USBH_MTP_OBJECT_PROP_BUFFER_SIZE,
    USBH_MTP_OBJECT_PROP_ENCODING_QUALITY,
    USBH_MTP_OBJECT_PROP_ENCODING_PROFILE,
    USBH_MTP_OBJECT_PROP_DISPLAY_NAME,
    USBH_MTP_OBJECT_PROP_BODY_TEXT,
    USBH_MTP_OBJECT_PROP_SUBJECT,
    USBH_MTP_OBJECT_PROP_PRIORITY,
    USBH_MTP_OBJECT_PROP_GIVEN_NAME,
    USBH_MTP_OBJECT_PROP_MIDDLE_NAMES,
    USBH_MTP_OBJECT_PROP_FAMILY_NAME,
    USBH_MTP_OBJECT_PROP_PREFIX,
    USBH_MTP_OBJECT_PROP_SUFFIX,
    USBH_MTP_OBJECT_PROP_PHONETIC_GIVEN_NAME,
    USBH_MTP_OBJECT_PROP_PHONETIC_FAMILY_NAME,
    USBH_MTP_OBJECT_PROP_EMAIL_PRIMARY,
    USBH_MTP_OBJECT_PROP_EMAIL_PERSONAL_1,
    USBH_MTP_OBJECT_PROP_EMAIL_PERSONAL_2,
    USBH_MTP_OBJECT_PROP_EMAIL_BUSINESS_1,
    USBH_MTP_OBJECT_PROP_EMAIL_BUSINESS_2,
    USBH_MTP_OBJECT_PROP_EMAIL_OTHERS,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_PRIMARY,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_PERSONAL,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_PERSONAL_2,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_BUSINESS,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_BUSINESS_2,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_MOBILE,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_MOBILE_2,
    USBH_MTP_OBJECT_PROP_FAX_NUMBER_PRIMARY,
    USBH_MTP_OBJECT_PROP_FAX_NUMBER_PERSONAL,
    USBH_MTP_OBJECT_PROP_FAX_NUMBER_BUSINESS,
    USBH_MTP_OBJECT_PROP_PAGER_NUMBER,
    USBH_MTP_OBJECT_PROP_PHONE_NUMBER_OTHERS,
    USBH_MTP_OBJECT_PROP_PRIMARY_WEB_ADDRESS,
    USBH_MTP_OBJECT_PROP_PERSONAL_WEB_ADDRESS,
    USBH_MTP_OBJECT_PROP_BUSINESS_WEB_ADDRESS,
    USBH_MTP_OBJECT_PROP_INSTANT_MESSENGER_ADDRESS,
    USBH_MTP_OBJECT_PROP_INSTANT_MESSENGER_ADDRESS_2,
    USBH_MTP_OBJECT_PROP_INSTANT_MESSENGER_ADDRESS_3,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_FULL,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_LINE_1,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_LINE_2,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_CITY,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_REGION,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_POSTAL_CODE,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_PERSONAL_COUNTRY,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_FULL,
    USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_LINE_1,
```

```
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_LINE_2,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_CITY,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_REGION,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_POSTAL_CODE,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_BUSINESS_COUNTRY,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_FULL,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_LINE_1,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_LINE_2,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_CITY,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_REGION,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_POSTAL_CODE,
  USBH_MTP_OBJECT_PROP_POSTAL_ADDRESS_OTHER_COUNTRY,
  USBH_MTP_OBJECT_PROP_ORGANIZATION_NAME,
  USBH_MTP_OBJECT_PROP_PHONETIC_ORGANIZATION_NAME,
  USBH_MTP_OBJECT_PROP_ROLE,
  USBH_MTP_OBJECT_PROP_BIRTHDATE,
  USBH_MTP_OBJECT_PROP_MESSAGE_TO,
  USBH_MTP_OBJECT_PROP_MESSAGE_CC,
  USBH_MTP_OBJECT_PROP_MESSAGE_BCC,
  USBH_MTP_OBJECT_PROP_MESSAGE_READ,
  USBH_MTP_OBJECT_PROP_MESSAGE_RECEIVED_TIME,
  USBH_MTP_OBJECT_PROP_MESSAGE_SENDER,
  USBH_MTP_OBJECT_PROP_ACTIVITY_BEGIN_TIME,
  USBH_MTP_OBJECT_PROP_ACTIVITY_END_TIME,
  USBH_MTP_OBJECT_PROP_ACTIVITY_LOCATION,
  USBH_MTP_OBJECT_PROP_ACTIVITY_REQUIRED_ATTENDEES,
  USBH_MTP_OBJECT_PROP_ACTIVITY_OPTIONAL_ATTENDEES,
  USBH_MTP_OBJECT_PROP_ACTIVITY_RESOURCES,
  USBH_MTP_OBJECT_PROP_ACTIVITY_ACCEPTED,
  USBH_MTP_OBJECT_PROP_OWNER,
  USBH_MTP_OBJECT_PROP_EDITOR,
  USBH_MTP_OBJECT_PROP_WEBMASTER,
  USBH_MTP_OBJECT_PROP_URL_SOURCE,
  USBH_MTP_OBJECT_PROP_URL_DESTINATION,
  USBH_MTP_OBJECT_PROP_TIME_BOOKMARK,
  USBH_MTP_OBJECT_PROP_OBJECT_BOOKMARK,
  USBH_MTP_OBJECT_PROP_BYTE_BOOKMARK,
  USBH_MTP_OBJECT_PROP_LAST_BUILD_DATE,
  USBH_MTP_OBJECT_PROP_TIME_TO_LIVE,
  USBH_MTP_OBJECT_PROP_MEDIA_GUID
} USBH_MTP_OBJECT_PROPERTIES;
```

### 9.5.0.3 USBH_MTP_RESPONSE_CODES

#### Description

Possible response codes reported by the device upon completion of an operation.

#### Type definition

```
typedef enum {
  USBH_MTP_RESPONSE_UNDEFINED,
  USBH_MTP_RESPONSE_OK,
  USBH_MTP_RESPONSE_GENERAL_ERROR,
  USBH_MTP_RESPONSE_PARAMETER_NOT_SUPPORTED,
  USBH_MTP_RESPONSE_INVALID_STORAGE_ID,
  USBH_MTP_RESPONSE_INVALID_OBJECT_HANDLE,
  USBH_MTP_RESPONSE_DEVICEPROP_NOT_SUPPORTED,
  USBH_MTP_RESPONSE_STORE_FULL,
  USBH_MTP_RESPONSE_STORE_NOT_AVAILABLE,
  USBH_MTP_RESPONSE_SPECIFICATION_BY_FORMAT_NOT_SUPPORTED,
  USBH_MTP_RESPONSE_NO_VALID_OBJECT_INFO,
  USBH_MTP_RESPONSE_DEVICE_BUSY,
  USBH_MTP_RESPONSE_INVALID_PARENT_OBJECT,
  USBH_MTP_RESPONSE_INVALID_PARAMETER,
  USBH_MTP_RESPONSE_SESSION_ALREADY_OPEN,
  USBH_MTP_RESPONSE_TRANSACTION_CANCELLED,
  USBH_MTP_RESPONSE_INVALID_OBJECT_PROP_CODE,
  USBH_MTP_RESPONSE_SPECIFICATION_BY_GROUP_UNSUPPORTED,
  USBH_MTP_RESPONSE_OBJECT_PROP_NOT_SUPPORTED
} USBH_MTP_RESPONSE_CODES;
```

## 9.5.0.4 USBH_MTP_OBJECT_FORMAT

### Description

Identifiers describing the format type of a given object.

### Type definition

```
typedef enum {
  USBH_MTP_OBJECT_FORMAT_UNDEFINED,
  USBH_MTP_OBJECT_FORMAT_ASSOCIATION,
  USBH_MTP_OBJECT_FORMAT_SCRIPT,
  USBH_MTP_OBJECT_FORMAT_EXECUTABLE,
  USBH_MTP_OBJECT_FORMAT_TEXT,
  USBH_MTP_OBJECT_FORMAT_HTML,
  USBH_MTP_OBJECT_FORMAT_DPOF,
  USBH_MTP_OBJECT_FORMAT_AIFF,
  USBH_MTP_OBJECT_FORMAT_WAV,
  USBH_MTP_OBJECT_FORMAT_MP3,
  USBH_MTP_OBJECT_FORMAT_AVI,
  USBH_MTP_OBJECT_FORMAT_MPEG,
  USBH_MTP_OBJECT_FORMAT_ASF,
  USBH_MTP_OBJECT_FORMAT_DEFINED,
  USBH_MTP_OBJECT_FORMAT_EXIF_JPEG,
  USBH_MTP_OBJECT_FORMAT_TIFF_EP,
  USBH_MTP_OBJECT_FORMAT_FLASHPIX,
  USBH_MTP_OBJECT_FORMAT_BMP,
  USBH_MTP_OBJECT_FORMAT_CIFF,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED2,
  USBH_MTP_OBJECT_FORMAT_GIF,
  USBH_MTP_OBJECT_FORMAT_JFIF,
  USBH_MTP_OBJECT_FORMAT_CD,
  USBH_MTP_OBJECT_FORMAT_PICT,
  USBH_MTP_OBJECT_FORMAT_PNG,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED3,
  USBH_MTP_OBJECT_FORMAT_TIFF,
  USBH_MTP_OBJECT_FORMAT_TIFF_IT,
  USBH_MTP_OBJECT_FORMAT_JP2,
  USBH_MTP_OBJECT_FORMAT_JPX,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_FIRMWARE,
  USBH_MTP_OBJECT_FORMAT_WINDOWS_IMAGE_FORMAT,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_AUDIO,
  USBH_MTP_OBJECT_FORMAT_WMA,
  USBH_MTP_OBJECT_FORMAT_OGG,
  USBH_MTP_OBJECT_FORMAT_AAC,
  USBH_MTP_OBJECT_FORMAT_AUDIBLE,
  USBH_MTP_OBJECT_FORMAT_FLAC,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_VIDEO,
  USBH_MTP_OBJECT_FORMAT_WMV,
  USBH_MTP_OBJECT_FORMAT_MP4_CONTAINER,
  USBH_MTP_OBJECT_FORMAT_MP2,
  USBH_MTP_OBJECT_FORMAT_3GP_CONTAINER,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_MULTIMEDIA_ALBUM,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_IMAGE_ALBUM,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_AUDIO_ALBUM,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_VIDEO_ALBUM,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_AUDIO_VIDEO_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_CONTACT_GROUP,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_MESSAGE_FOLDER,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_CHAPTERED_PRODUCTION,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_AUDIO_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_VIDEO_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_MEDIACAST,
  USBH_MTP_OBJECT_FORMAT_WPL_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_M3U_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_MPL_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_ASX_PLAYLIST,
```

```
  USBH_MTP_OBJECT_FORMAT_PLS_PLAYLIST,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_DOCUMENT,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_DOCUMENT,
  USBH_MTP_OBJECT_FORMAT_XML_DOCUMENT,
  USBH_MTP_OBJECT_FORMAT_MICROSOFT_WORD_DOCUMENT,
  USBH_MTP_OBJECT_FORMAT_MHT_COMPILED_HTML_DOCUMENT,
  USBH_MTP_OBJECT_FORMAT_MICROSOFT_EXCEL_SPREADSHEET,
  USBH_MTP_OBJECT_FORMAT_MICROSOFT_POWERPOINT_PRESENTATION,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_MESSAGE,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_MESSAGE,
  USBH_MTP_OBJECT_FORMAT_UNDEFINED_CONTACT,
  USBH_MTP_OBJECT_FORMAT_ABSTRACT_CONTACT,
  USBH_MTP_OBJECT_FORMAT_VCARD_2
} USBH_MTP_OBJECT_FORMAT;
```

### 9.5.0.5 USBH_MTP_EVENT_CODES

#### Description

Events are described by a 16-bit code.

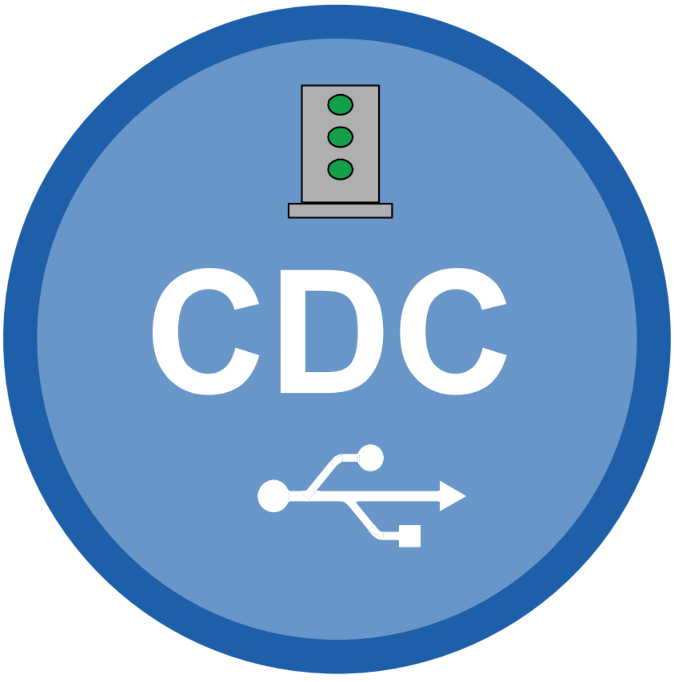
#### Type definition

```
typedef enum {
  USBH_MTP_EVENT_UNDEFINED,
  USBH_MTP_EVENT_CANCEL_TRANSACTION,
  USBH_MTP_EVENT_OBJECT_ADDED,
  USBH_MTP_EVENT_OBJECT_REMOVED,
  USBH_MTP_EVENT_STORE_ADDED,
  USBH_MTP_EVENT_STORE_REMOVED,
  USBH_MTP_EVENT_DEVICE_PROP_CHANGED,
  USBH_MTP_EVENT_OBJECT_INFO_CHANGED,
  USBH_MTP_EVENT_DEVICE_INFO_CHANGED,
  USBH_MTP_EVENT_REQUEST_OBJECT_TRANSFER,
  USBH_MTP_EVENT_STORE_FULL,
  USBH_MTP_EVENT_DEVICE_RESET,
  USBH_MTP_EVENT_STORAGE_INFO_CHANGED,
  USBH_MTP_EVENT_CAPTURE_COMPLETE,
  USBH_MTP_EVENT_UNREPORTED_STATUS,
  USBH_MTP_EVENT_OBJECT_PROP_CHANGED,
  USBH_MTP_EVENT_OBJECT_PROP_DESC_CHANGED,
  USBH_MTP_EVENT_OBJECT_REFERENCES_CHANGED
} USBH_MTP_EVENT_CODES;
```

# Chapter 10

# 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.



# 10.1 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 'USBH_CDC_'.

## 10.1.1 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.

## 10.1.2 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.

## 10.1.3 Example code

An example application which uses the API is provided in the `USBH_CDC_Start.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.

# 10.2 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 and registers the CDC device module with emUSB-Host. |
| USBH_CDC_Exit() | Unregisters and de-initializes the CDC device module from emUSB-Host. |
| USBH_CDC_AddNotification() | Adds a callback in order to be notified when a device is added or removed. |
| USBH_CDC_RemoveNotification() | Removes a callback added via USBH_CDC_AddNotification. |
| USBH_CDC_RegisterNotification() | This function is deprecated, please use function USBH_CDC_AddNotification! Sets a callback in order to be notified when a device is added or removed. |
| USBH_CDC_ConfigureDefaultTimeout() | Sets the default read and write time-out 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() | Enables or disables short read mode. |
| USBH_CDC_GetDeviceInfo() | Retrieves information about the CDC device. |
| USBH_CDC_SetTimeouts() | Sets up the timeouts for read and write operations. |
| USBH_CDC_Read() | Reads from the CDC device. |
| USBH_CDC_Write() | Writes data to the CDC device. |
| USBH_CDC_ReadAsync() | Triggers a read transfer to the CDC device. |
| USBH_CDC_WriteAsync() | Triggers a write transfer to the CDC device. |
| USBH_CDC_CancelRead() | Cancels a running read transfer. |
| USBH_CDC_CancelWrite() | Cancels a running write transfer. |
| 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 limited time. |
| USBH_CDC_SetBreakOn() | Sets the BREAK condition for the device to "on". |

| Function | Description |
|---|---|
| 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_SetOnSerialStateChange() | Sets a callback which informs the user about serial state changes. |
| USBH_CDC_SetOnIntStateChange() | Sets the callback to retrieve data that are received on the interrupt endpoint. |
| USBH_CDC_GetSerialNumber() | Get the serial number of a CDC device. |
| USBH_CDC_AddDevice() | Register a device with a non-standard interface layout as a CDC device. |
| USBH_CDC_RemoveDevice() | Removes a non-standard CDC device which was added by USBH_CDC_AddDevice(). |
| USBH_CDC_SetConfigFlags() | Sets configuration flags for the CDC module. |
| USBH_CDC_SuspendResume() | Prepares a CDC device for suspend (stops the interrupt endpoint) or re-starts the interrupt endpoint functionality after a resume. |

## 10.2.1 USBH_CDC_Init()

### Description

Initializes and registers the CDC device module with emUSB-Host.

### Prototype

```
U8 USBH_CDC_Init(void);
```

### Return value

| | |
|---|---|
| 1 | Success or module already initialized. |
| 0 | Could not register CDC device module. |

### Additional information

This function can be called multiple times, but only the first call initializes the module. Any further calls only increase the initialization counter. This is useful for cases where the module is initialized from different places which do not interact with each other, To de-initialize the module `USBH_CDC_Exit` has to be called the same number of times as this function was called.

## 10.2.2 USBH_CDC_Exit()

### Description

Unregisters and de-initializes the CDC device module from emUSB-Host.

### Prototype

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

### Additional information

Has to be called the same number of times USBH_CDC_Init was called in order to de-initialize the module. 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.

# 10.2.3 USBH_CDC_AddNotification()

### Description

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

### Prototype

```
USBH_STATUS USBH_CDC_AddNotification(USBH_NOTIFICATION_HOOK * pHook,
                                     USBH_NOTIFICATION_FUNC * pfNotification,
                                     void                   * pContext);
```

### Parameters

| Parameter | Description |
|---|---|
| pHook | Pointer to a user provided USBH_NOTIFICATION_HOOK variable. |
| pfNotification | Pointer to a function the stack should call when a device is connected or disconnected. |
| pContext | Pointer to a user context that is passed to the callback function. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Example

```
static USBH_NOTIFICATION_HOOK _Hook;

/*********************************************************************
*
*       _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) {
  (void)pContext;
  switch (Event) {
  case USBH_DEVICE_EVENT_ADD:
    USBH_Logf_Application("**** Device added\n");
    _DevIndex = DevIndex;
    _DevIsReady = 1;
    break;
  case USBH_DEVICE_EVENT_REMOVE:
    USBH_Logf_Application("**** Device removed\n");
    _DevIsReady = 0;
    _DevIndex   = -1;
    break;
  default:;   // Should never happen
  }
}

<...>
USBH_CDC_Init();
USBH_CDC_AddNotification(&_Hook, _cbOnAddRemoveDevice, NULL);
<...>
```

# 10.2.4 USBH_CDC_RemoveNotification()

### Description

Removes a callback added via `USBH_CDC_AddNotification`.

### Prototype

```
USBH_STATUS USBH_CDC_RemoveNotification(const USBH_NOTIFICATION_HOOK * pHook);
```

### Parameters

| Parameter | Description |
|---|---|
| pHook | Pointer to a user provided `USBH_NOTIFICATION_HOOK` variable. |

### Return value

`USBH_STATUS_SUCCESS` on success or error code on failure.

## 10.2.5 USBH_CDC_RegisterNotification()

### Description

This function is deprecated, please use function USBH_CDC_AddNotification! 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);
```

### Parameters

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

### Additional information

This function is deprecated, please use function USBH_CDC_AddNotification.

# 10.2.6 USBH_CDC_ConfigureDefaultTimeout()

## Description

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

## Prototype

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

## Parameters

| Parameter | Description |
|---|---|
| ReadTimeout | Default read timeout given in ms. |
| WriteTimeout | Default write timeout given in ms. |

## 10.2.7 USBH_CDC_Open()

### Description

Opens a device given by an index.

### Prototype

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

### Parameters

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

### Return value

| | |
| --- | --- |
| ≠ 0 | Handle to a CDC device |
| = 0 | Device not available or removed. |

### Additional information

The index of a new connected device is provided to the callback function registered with `USBH_CDC_RegisterNotification()`.

## 10.2.8 USBH_CDC_Close()

### Description

Closes a handle to an opened device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.9 USBH_CDC_AllowShortRead()

### Description

Enables or disables short read mode. If enabled, the function USBH_CDC_Read() 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.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| AllowShortRead | Define whether short read mode shall be used or not.<br>• 1 - Allow short read.<br>• 0 - Short read mode disabled. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.10 USBH_CDC_GetDeviceInfo()

### Description

Retrieves information about the CDC device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pDevInfo | Pointer to a USBH_CDC_DEVICE_INFO structure that receives the information. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.11 USBH_CDC_SetTimeouts()

### Description

Sets up the timeouts for read and write operations.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| ReadTimeout | Read timeout given in ms. |
| WriteTimeout | Write timeout given in ms. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.12 USBH_CDC_Read()

### Description

Reads from the CDC device. Depending of the ShortRead mode (see USBH_CDC_AllowShortRead()), this function will either return as soon as data are available or all data have been read from the device. This function will also return when a set timeout is expired, whatever comes first. If a timeout is not specified via USBH_CDC_SetTimeouts() the default timeout (USBH_CDC_DEFAULT_TIMEOUT) is used.

The USB stack can only read complete packets from the USB device. If the size of a received packet exceeds NumBytes than all data that does not fit into the callers buffer (pData) is stored in an internal buffer and will be returned by the next call to USBH_CDC_Read(). See also USBH_CDC_GetQueueStatus().

To read a null packet, set pData = NULL and NumBytes = 0.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pData | Pointer to a buffer to store the read data. |
| NumBytes | Number of bytes to be read from the device. |
| pNumBytesRead | Pointer to a variable which receives the number of bytes read from the device. Can be NULL. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

If the function returns an error code (including USBH_STATUS_TIMEOUT) it already may have read part of the data. The number of bytes read successfully is always stored in the variable pointed to by pNumBytesRead.

# 10.2.13 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 (USBH_CDC_DEFAULT_TIMEOUT) is used.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pData | Pointer to data to be sent. |
| NumBytes | Number of bytes to send. |
| pNumBytesWritten | Pointer to a variable which receives the number of bytes written to the device. Can be NULL. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

If the function returns an error code (including USBH_STATUS_TIMEOUT) it already may have written part of the data. The number of bytes written successfully is always stored in the variable pointed to by pNumBytesWritten.

## 10.2.14 USBH_CDC_ReadAsync()

### Description

Triggers a read transfer to the CDC device. The result of the transfer is received through the user callback. This function will return immediately while the read transfer is done asynchronously. The read operation terminates either, if 'BuffSize' bytes have been read or if a short packet was received from the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pBuffer | Pointer to the buffer that receives the data from the device. |
| BufferSize | Size of the buffer in bytes. Must be a multiple of of the maximum packet size of the USB device. |
| pfOnComplete | Pointer to a user function of type USBH_CDC_ON_COMPLETE_FUNC which will be called after the transfer has been completed. |
| pRWContext | 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. The member 'pUserContext' may be set before calling USBH_CDC_ReadAsync(). Other members need not be initialized and are set by the function USBH_CDC_ReadAsync(). The memory used for this structure must be valid, until the transaction is completed. |

### Return value

| | |
|---|---|
| = USBH_STATUS_PENDING | Success, the data transfer is queued, the user callback will be called after the transfer is finished. |
| ≠ USBH_STATUS_PENDING | An error occurred, the transfer is not started and user callback will not be called. |

### Additional information

This function performs an unbuffered read operation (in contrast to USBH_CDC_Read()), so care should be taken if intermixing calls to USBH_CDC_ReadAsync() and USBH_CDC_Read().

### Example

```
static USBH_CDC_RW_CONTEXT _ReadWriteContext;

<...>

/*********************************************************************
*
*       _OnReadComplete
*/
static void _OnReadComplete(USBH_CDC_RW_CONTEXT * pRWContext) {
  if (pRWContext->Status == USBH_STATUS_SUCCESS) {
    printf("Successfully read %u bytes \n",
```

```
              (unsigned int)pRWContext->NumBytesTransferred);
  } else {
    printf("ReadAsync callback returned %s \n",
            USBH_GetStatusStr(pRWContext->Status));
    // Error handling
  }
  <...>
}

<...>

Status = USBH_CDC_ReadAsync(_hDevice,
                            _acBuffer,
                            NumBytes,
                            _OnReadComplete,
                            &_ReadWriteContext);
if (Status != USBH_STATUS_PENDING) {
  // Error handling.
}
<...>
```

# 10.2.15 USBH_CDC_WriteAsync()

### Description

Triggers a write transfer to the CDC device. The result of the transfer is received through the user callback. This function will return immediately while the write transfer is done asynchronously.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pBuffer | Pointer to a buffer which holds the data. |
| BufferSize | Number of bytes to write. |
| pfOnComplete | Pointer to a user function of type USBH_CDC_ON_COMPLETE_FUNC which will be called after the transfer has been completed. |
| pRWContext | 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. pfOnComplete function. The member 'pUserContext' may be set before calling USBH_CDC_WriteAsync(). Other members need not be initialized and are set by the function USBH_CDC_WriteAsync(). The memory used for this structure must be valid, until the transaction is completed. |

### Return value

| | |
|---|---|
| = USBH_STATUS_PENDING | Success, the data transfer is queued, the user callback will be called after the transfer is finished. |
| ≠ USBH_STATUS_PENDING | An error occurred, the transfer is not started and user callback will not be called. |

### Example

```
static USBH_CDC_RW_CONTEXT _ReadWriteContext;

<...>

/*********************************************************************
*
*       _OnWriteComplete
*/
static void _OnWriteComplete(USBH_CDC_RW_CONTEXT * pRWContext) {
  if (pRWContext->Status == USBH_STATUS_SUCCESS) {
    printf("Successfully written data to the device \n");
  } else {
    printf("WriteAsync callback returned %s \n",
          USBH_GetStatusStr(pRWContext->Status));
    // Error handling
  }
  <...>
}
```

```
<...>


Status = USBH_CDC_WriteAsync(_hDevice,
                             _acBuffer,
                             NumBytes,
                             _OnWriteComplete,
                             &_ReadWriteContext);
if (Status != USBH_STATUS_PENDING) {
  // Error handling.
}
<...>
```

# 10.2.16 USBH_CDC_CancelRead()

### Description

Cancels a running read transfer.

### Prototype

```
USBH_STATUS USBH_CDC_CancelRead(USBH_CDC_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

This function can be used to cancel a transfer which was initiated by USBH_CDC_ReadAsync or USBH_CDC_Read. In the later case this function has to be called from a different task.

## 10.2.17 USBH_CDC_CancelWrite()

### Description

Cancels a running write transfer.

### Prototype

```
USBH_STATUS USBH_CDC_CancelWrite(USBH_CDC_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

This function can be used to cancel a transfer which was initiated by USBH_CDC_WriteAsync or USBH_CDC_Write. In the later case this function has to be called from a different task.

# 10.2.18 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);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| Baudrate | Transfer rate. |
| DataBits | Number of bits per word. Must be between USBH_CD-C_BITS_5 and USBH_CDC_BITS_8. |
| StopBits | Number of stop bits. Must be USBH_CDC_STOP_BITS_1 or USBH_CDC_STOP_BITS_2. |
| Parity | Parity - must be must be one of the following values:<br>• UBSH_CDC_PARITY_NONE<br>• UBSH_CDC_PARITY_ODD<br>• UBSH_CDC_PARITY_EVEN<br>• UBSH_CDC_PARITY_MARK<br>• USBH_CDC_PARITY_SPACE |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.19 USBH_CDC_SetDtr()

### Description

Sets the Data Terminal Ready (DTR) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.20 USBH_CDC_ClrDtr()

### Description

Clears the Data Terminal Ready (DTR) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.21 USBH_CDC_SetRts()

### Description

Sets the Request To Send (RTS) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.22 USBH_CDC_ClrRts()

### Description

Clears the Request To Send (RTS) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by `USBH_CDC_Open()`. |

### Return value

`USBH_STATUS_SUCCESS` on success or error code on failure.

## 10.2.23 USBH_CDC_GetQueueStatus()

### Description

Gets the number of bytes in the receive queue.

The USB stack can only read complete packets from the USB device. If the size of a received packet exceeds the number of bytes requested with `USBH_CDC_Read()`, than all data that is not returned by `USBH_CDC_Read()` is stored in an internal buffer.

The number of bytes returned by `USBH_CDC_GetQueueStatus()` can be read using `USBH_CDC_Read()` out of the buffer without a USB transaction to the USB device being executed.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by `USBH_CDC_Open()`. |
| pRxBytes | Pointer to a variable which receives the number of bytes in the receive queue. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### 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.
  //
  if (USBH_CDC_GetQueueStatus(hDevice, &RxBytes) == USBH_STATUS_SUCCESS) {
    //
    // Read the remaining bytes.
    //
    if (RxBytes > 0) {
      USBH_CDC_Read(hDevice, &acData[1], RxBytes, &NumBytes);
    }
  }
}
```

## 10.2.24 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);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| Duration | Duration of the break condition in ms. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.25 USBH_CDC_SetBreakOn()

### Description

Sets the BREAK condition for the device to “on”.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 10.2.26 USBH_CDC_SetBreakOff()

### Description

Resets the BREAK condition for the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.27 USBH_CDC_GetSerialState()

### Description

Gets the modem status and line status from the device.

### Prototype

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

### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pSerialState | Pointer to a structure of type USBH_CDC_SERIALSTATE which receives the serial status from the device. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### 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

## 10.2.28 USBH_CDC_SetOnSerialStateChange()

### Description

Sets a callback which informs the user about serial state changes.

### Prototype

```
USBH_STATUS USBH_CDC_SetOnSerialStateChange
                          (USBH_CDC_HANDLE                  hDevice,
                           USBH_CDC_SERIAL_STATE_CALLBACK * pfOnSerialStateChange);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pfOnSerialStateChange | Pointer to the user callback. Can be NULL (to remove the callback). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

The callback is called in the context of the ISR task. The callback should not block.

## 10.2.29 USBH_CDC_SetOnIntStateChange()

### Description

Sets the callback to retrieve data that are received on the interrupt endpoint.

### Prototype

```
USBH_STATUS USBH_CDC_SetOnIntStateChange
                                          (USBH_CDC_HANDLE                hDevice,
                                           USBH_CDC_INT_STATE_CALLBACK * pfOnIntState,
                                           void                         * pUserContext);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| pfOnIntState | Pointer to the callback that shall retrieve the data. |
| pUserContext | Pointer to the user context. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.30 USBH_CDC_GetSerialNumber()

### Description

Get the serial number of a CDC device. The serial number is in UNICODE format, not zero terminated.

### Prototype

```
USBH_STATUS USBH_CDC_GetSerialNumber(USBH_CDC_HANDLE   hDevice,
                                     U32               BuffSize,
                                     U8              * pSerialNumber,
                                     U32             * pSerialNumberSize);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| BuffSize | Pointer to a buffer which holds the data. |
| pSerialNumber | Size of the buffer in bytes. |
| pSerialNumberSize | Pointer to a user function which will be called. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.31 USBH_CDC_AddDevice()

### Description

Register a device with a non-standard interface layout as a CDC device. This function should not be used for CDC compliant devices! 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 ControlInterfaceID,
                               USBH_INTERFACE_ID DataInterfaceId,
                               unsigned          Flags);
```

### Parameters

| Parameter | Description |
|---|---|
| ControlInterfaceID | Numeric index of the CDC ACM interface. |
| DataInterfaceId | Numeric index of the CDC Data interface. |
| Flags | Reserved for future use. Should be zero. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

The numeric interface IDs can be retrieved 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.

## 10.2.32 USBH_CDC_RemoveDevice()

### Description

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

### Prototype

```
USBH_STATUS USBH_CDC_RemoveDevice(USBH_INTERFACE_ID ControlInterfaceID,
                                  USBH_INTERFACE_ID DataInterfaceId);
```

### Parameters

| Parameter | Description |
|---|---|
| ControlInterfaceID | Numeric index of the CDC ACM interface. |
| DataInterfaceId | Numeric index of the CDC Data interface. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

# 10.2.33 USBH_CDC_SetConfigFlags()

### Description

Sets configuration flags for the CDC module.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| Flags | A bitwise OR-combination of flags that shall be set for each device. At the moment the following are available:<br>• USBH_CDC_IGNORE_INT_EP<br>• USBH_CDC_DISABLE_INTERFACE_CHECK. |

## 10.2.34 USBH_CDC_SuspendResume()

### Description

Prepares a CDC device for suspend (stops the interrupt endpoint) or re-starts the interrupt endpoint functionality after a resume.

### Prototype

```
USBH_STATUS USBH_CDC_SuspendResume(USBH_CDC_HANDLE hDevice,
                                   U8              State);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_CDC_Open(). |
| State | 0 - Prepare for suspend. 1 - Return from resume. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

The application must make sure no transactions are running when setting a device into suspend mode. This function is used in combination with USBH_SetRootPortPower(). Call this function before USBH_SetRootPortPower(x, y, USBH_SUSPEND) with State = 0. Call this function after USBH_SetRootPortPower(x, y, USBH_NORMAL_POWER) with State = 1;

# 10.3 Data structures

This chapter describes the emUSB-Host CDC driver data structures.

| Structure | Description |
|---|---|
| USBH_CDC_DEVICE_INFO | Structure containing information about a CDC device. |
| USBH_CDC_SERIALSTATE | Structure describing the serial state of CDC device. |
| USBH_CDC_RW_CONTEXT | Contains information about a completed, asynchronous transfers. |

## 10.3.1 USBH_CDC_DEVICE_INFO

### Description

Structure containing information about a CDC device.

### Type definition

```
typedef struct {
  U16                VendorId;
  U16                ProductId;
  USBH_SPEED         Speed;
  U8                 ControlInterfaceNo;
  U8                 DataInterfaceNo;
  U16                MaxPacketSize;
  U16                ControlClass;
  U16                ControlSubClass;
  U16                ControlProtocol;
  U16                DataClass;
  U16                DataSubClass;
  U16                DataProtocol;
  USBH_INTERFACE_ID  ControlInterfaceID;
  USBH_INTERFACE_ID  DataInterfaceID;
} USBH_CDC_DEVICE_INFO;
```

### Structure members

| Member | Description |
|---|---|
| VendorId | The Vendor ID of the device. |
| ProductId | The Product ID of the device. |
| Speed | The USB speed of the device, see USBH_SPEED. |
| ControlInterfaceNo | Interface index of the ACM Control interface (from USB descriptor). |
| DataInterfaceNo | Interface index of the ACM Data interface (from USB descriptor). |
| MaxPacketSize | Maximum packet size of the device, usually 64 in full-speed and 512 in high-speed. |
| ControlClass | The Class value field of the control interface |
| ControlSubClass | The SubClass value field of the control interface |
| ControlProtocol | The Protocol value field of the control interface |
| DataClass | The Class value field of the data interface |
| DataSubClass | The SubClass value field of the data interface |
| DataProtocol | The Protocol value field of the data interface |
| ControlInterfaceID | ID of the ACM control interface. |
| DataInterfaceID | ID of the ACM data interface. |

## 10.3.2 USBH_CDC_SERIALSTATE

### Description

Structure describing the serial state of CDC device. All members can have a value of 0 (= false/off) or 1 (= true/on).

### Type definition

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

### Structure 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. |

## 10.3.3 USBH_CDC_RW_CONTEXT

### Description

Contains information about a completed, asynchronous transfers. Is passed to the USBH_CDC_ON_COMPLETE_FUNC user callback when using asynchronous write and read. When this structure is passed to USBH_CDC_ReadAsync() or USBH_CDC_WriteAsync() its member need not to be initialized.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| pUserContext | Pointer to a user context. Can be arbitrarily used by the application. |
| Status | Result status of the asynchronous transfer. |
| NumBytesTransferred | Number of bytes transferred. |
| pUserBuffer | Pointer to the buffer provided to USBH_CDC_ReadAsync() or USBH_CDC_WriteAsync(). |
| UserBufferSize | Size of the buffer as provided to USBH_CDC_ReadAsync() or USBH_CDC_WriteAsync(). |

# 10.4 Type definitions

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

| Type | Description |
|---|---|
| USBH_CDC_ON_COMPLETE_FUNC | Function called on completion of an asynchronous transfer. |
| USBH_CDC_SERIAL_STATE_CALLBACK | Function called on a reception of a CDC ACM serial state change. |

## 10.4.1 USBH_CDC_ON_COMPLETE_FUNC

### Description

Function called on completion of an asynchronous transfer. Used by the functions `USBH_CDC_ReadAsync()` and `USBH_CDC_WriteAsync()`.

### Type definition

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

### Parameters

| Parameter | Description |
|---|---|
| pRWContext | Pointer to a USBH_CDC_RW_CONTEXT structure. |

## 10.4.2 USBH_CDC_SERIAL_STATE_CALLBACK

### Description

Function called on a reception of a CDC ACM serial state change. Used by the function `USBH_CDC_SetOnSerialStateChange()`.

### Type definition

```
typedef void USBH_CDC_SERIAL_STATE_CALLBACK(USBH_CDC_HANDLE        hDevice,
                                             USBH_CDC_SERIALSTATE * pSerialState);
```

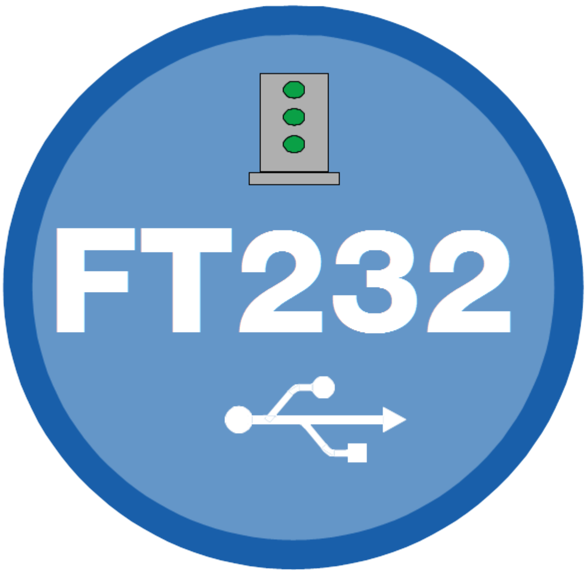
### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to an open device returned by `USBH_CDC_Open()`. |
| pSerialState | Pointer to a structure of type `USBH_CDC_SERIALSTATE` showing the serial status from the device. |

# Chapter 11

# 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.



# 11.1 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.

## 11.1.1 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.

## 11.1.2 Example code

An example application which uses the API is provided in the USBH_FT232_Start.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.

## 11.1.3 Compatibility

The following devices work with the current FT232 driver:

- FT8U232AM
- FT232B
- FT232R
- FT2232D

## 11.1.4 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.

## 11.2 API Functions

This chapter describes the emUSB-Host FT232 driver API functions.

| Function | Description |
|---|---|
| USBH_FT232_Init() | Initializes and registers the FT232 device driver with emUSB-Host. |
| USBH_FT232_Exit() | Unregisters and de-initializes the FT232 device driver from emUSB-Host. |
| 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 timeouts the host waits until the data transfer will be aborted for a specific FT232 device. |
| 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() | This function 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. |

| Function | Description |
|---|---|
| USBH_FT232_SetLatencyTimer() | The latency timer controls the timeout for the FTDI device to transfer data from the FT232 interface to the USB interface. |
| USBH_FT232_GetLatencyTimer() | Get the current value of the latency timer. |
| USBH_FT232_SetBitMode() | Enables different chip modes. |
| USBH_FT232_GetBitMode() | Returns the current values on the data bus pins. |

## 11.2.1 USBH_FT232_Init()

### Description

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

### Prototype

```
U8 USBH_FT232_Init(void);
```

### Return value

| | |
|---|---|
| 1 | Success. |
| 0 | Could not register FT232 device driver. |

# 11.2.2 USBH_FT232_Exit()

## Description

Unregisters and de-initializes 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.

## 11.2.3 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);
```

### Parameters

| 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. |

### Additional information

Only one notification function can be set for all FT232 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:
    USBH_Logf_Application("**** Device added\n");
    _DevIndex = DevIndex;
    _DevIsReady = 1;
    break;
  case USBH_DEVICE_EVENT_REMOVE:
    USBH_Logf_Application("**** Device removed\n");
    _DevIsReady = 0;
    _DevIndex   = -1;
    break;
  default:;   // Should never happen
  }
}

<...>
USBH_FT232_Init();
USBH_FT232_RegisterNotification(_cbOnAddRemoveDevice, NULL);
<...>
```

# 11.2.4 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);
```

### Parameters

| Parameter | Description |
|---|---|
| ReadTimeout | Default read timeout given in ms. |
| WriteTimeout | Default write timeout given in ms. |

### Additional information

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

## 11.2.5 USBH_FT232_Open()

### Description

Opens a device given by an index.

### Prototype

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

### Parameters

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

### Return value

≠ 0 Handle to the device.
= 0 Device could not be opened (removed or not available).

## 11.2.6 USBH_FT232_Close()

### Description

Closes a handle to an opened device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to a opened device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.7 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);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pDevInfo | out Pointer to a USBH_FT232_DEVICE_INFO structure to store information related to the device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.8 USBH_FT232_ResetDevice()

### Description

Resets the FT232 device

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.9 USBH_FT232_SetTimeouts()

### Description

Sets up the timeouts the host waits until the data transfer will be aborted for a specific FT232 device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| ReadTimeout | Read time-out given in ms. |
| WriteTimeout | Write time-out given in ms. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.10 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);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pData | Pointer to a buffer to store the read data. |
| NumBytes | Number of bytes to be read from the device. |
| pNumBytesRead | out Pointer to a variable which receives the number of bytes read from the device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

### 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 data 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.

## 11.2.11 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);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pData | in Pointer to data to be sent. |
| NumBytes | Number of bytes to write to the device. |
| pNumBytesWritten | out Pointer to a variable which receives the number of bytes written to the device. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 11.2.12 USBH_FT232_AllowShortRead()

### Description

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

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| AllowShortRead | Define whether short read mode shall be used or not.<br>1 - Allow short read.<br>0 - Short read mode disabled. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### 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 any 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 should be set to 0.

## 11.2.13 USBH_FT232_SetBaudRate()

### Description

Sets the baud rate for the opened device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| BaudRate | Baudrate to set. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.14 USBH_FT232_SetDataCharacteristics()

### Description

Setups the serial communication with the given characteristics.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| Length | Number of bits per word. Must be either USBH_FT232_BITS_8 or USBH_FT232_BITS_7. |
| StopBits | Number of stop bits. Must be USBH_FT232_STOP_BITS_1 or USBH_FT232_STOP_BITS_2. |
| Parity | Parity - must be one of the following values:<br>USBH_FT232_PARITY_NONE<br>USBH_FT232_PARITY_ODD<br>USBH_FT232_PARITY_EVEN<br>USBH_FT232_PARITY_MARK<br>USBH_FT232_PARITY_SPACE |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.15 USBH_FT232_SetFlowControl()

### Description

This function sets the flow control for the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| FlowControl | Must be one of the following values:<br>USBH_FT232_FLOW_NONE<br>USBH_FT232_FLOW_RTS_CTS<br>USBH_FT232_FLOW_DTR_DSR<br>USBH_FT232_FLOW_XON_XOFF |
| XonChar | Character used to signal Xon. Only used if flow control is FT_FLOW_XON_XOFF. |
| XoffChar | Character used to signal Xoff. Only used if flow control is FT_FLOW_XON_XOFF. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.2.16 USBH_FT232_SetDtr()

## Description

Sets the Data Terminal Ready (DTR) control signal.

## Prototype

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

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.2.17 USBH_FT232_ClrDtr()

### Description

Clears the Data Terminal Ready (DTR) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 11.2.18 USBH_FT232_SetRts()

### Description

Sets the Request To Send (RTS) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

= USBH_STATUS_SUCCESS   Successful.
≠ USBH_STATUS_SUCCESS   An error occurred.

# 11.2.19 USBH_FT232_ClrRts()

### Description

Clears the Request To Send (RTS) control signal.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

# 11.2.20 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);
```

## Parameters

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

## Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

## 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

# 11.2.21 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);
```

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| EventChar | Eventc character. |
| EventCharEnabled | 0, if event character disabled, non-zero otherwise. |
| ErrorChar | Error character. |
| ErrorCharEnabled | 0, if error character disabled, non-zero otherwise. |

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## Additional information

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

# 11.2.22 USBH_FT232_Purge()

### Description

Purges receive and transmit buffers in the device.

### Prototype

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

### Parameters

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

### Return value

| | |
| --- | --- |
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.2.23 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);
```

## Parameters

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

## Return value

| | |
| --- | --- |
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.2.24 USBH_FT232_SetBreakOn()

### Description

Sets the BREAK condition for the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.2.25 USBH_FT232_SetBreakOff()

### Description

Resets the BREAK condition for the device.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

# 11.2.26 USBH_FT232_SetLatencyTimer()

## Description

The latency timer controls the timeout for the FTDI device to transfer data from the FT232 interface to the USB interface. The FTDI device transfers data from the FT232 to the USB interface when it receives 62 bytes over FT232 (one full packet with 2 status bytes) or when the latency timeout elapses.

## Prototype

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

## Parameters

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

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

## 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.

## 11.2.27 USBH_FT232_GetLatencyTimer()

### Description

Get the current value of the latency timer.

### Prototype

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

### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to the opened device. |
| pLatency | Pointer to a value which receives the device latency setting. |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Additional information

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

# 11.2.28 USBH_FT232_SetBitMode()

### Description

Enables different chip modes.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| Mask | 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. |
| Enable | Mode value. Can be one of the following values:<br>• 0x00 = Reset<br>• 0x01 = Asynchronous Bit Bang<br>• 0x02 = MPSSE (FT2232, FT2232H, FT4232H and FT232H devices only)<br>• 0x04 = Synchronous Bit Bang (FT232R, FT245R, FT2232, FT2232H, FT4232H and FT232H devices only)<br>• 0x08 = MCU Host Bus Emulation Mode (FT2232, FT2232H, FT4232H and FT232H devices only)<br>• 0x10 = Fast Opto-Isolated Serial Mode (FT2232, FT2232H, FT4232H and FT232H devices only)<br>• 0x20 = CBUS Bit Bang Mode (FT232R and FT232H devices only)<br>• 0x40 = Single Channel Synchronous 245 FIFO Mode (FT2232H and FT232H devices only). |

### Return value

= USBH_STATUS_SUCCESS Successful.
≠ USBH_STATUS_SUCCESS An error occurred.

### Additional information

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

# 11.2.29 USBH_FT232_GetBitMode()

## Description

Returns the current values on the data bus pins. This function does NOT return the configured mode.

## Prototype

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

## Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to the opened device. |
| pMode | Pointer to a U8 variable to store the current value. |

## Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Successful. |
| ≠ USBH_STATUS_SUCCESS | An error occurred. |

# 11.3 Data structures

This chapter describes the emUSB-Host FT232 driver data structures.

| Function | Description |
|---|---|
| USBH_FT232_DEVICE_INFO | Contains information about an FT232 device. |

## 11.3.1 USBH_FT232_DEVICE_INFO

### Description

Contains information about an FT232 device.

### Type definition

```
typedef struct {
  U16         VendorId;
  U16         ProductId;
  U16         bcdDevice;
  USBH_SPEED  Speed;
  U16         MaxPacketSize;
} USBH_FT232_DEVICE_INFO;
```

### Structure members

| Member | Description |
| --- | --- |
| VendorId | USB Vendor Id. |
| ProductId | USB Product Id. |
| bcdDevice | The BCD coded device version. |
| Speed | Operation speed of the device. See USBH_SPEED. |
| MaxPacketSize | Maximum size of a single packet in bytes. |

# Chapter 12

# BULK Device Driver (Add-On)

This chapter describes the optional emUSB-Host add-on “BULK device driver”. It allows communication with a vendor specific USB devices.

# 12.1 Introduction

The BULK driver software component of emUSB-Host allows communication with vendor specific devices using an arbitrary number of bulk or interrupt endpoints.

This chapter provides an explanation of the functions available to application developers via the BULK driver software. All the functions and data types of this add-on are prefixed with 'USBH_BULK_'.

## 12.1.1 Overview

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

## 12.1.2 Features

The following features are provided:

- Ability to send and receive data.
- Handling of multiple BULK devices at the same time.
- Notifications about BULK connection status.
- Handling for an arbitrary number of endpoints.

## 12.1.3 Example code

An example application which uses the API is provided in the `USBH_BULK_Start.c` file. This example demonstrates simple communication between the host and a Bulk device. To run this sample a device programmed with the emUSB-Device sample `USB_BULK_Test.c` is required. The sample demonstrates how to extract the endpoint addresses, which are required by the emUSB-Host BULK API. The sample will send and receive data starting with 1 byte, after each successful echo the number of bytes is increased, up to 1024.

## 12.2 API Functions

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

| Function | Description |
|---|---|
| USBH_BULK_Init() | Initializes and registers the BULK device module with emUSB-Host. |
| USBH_BULK_Exit() | Unregisters and de-initializes the BULK device module from emUSB-Host. |
| USBH_BULK_RegisterNotification() | (Deprecated) Sets a callback in order to be notified when a device is added or removed. |
| USBH_BULK_AddNotification() | Adds a callback in order to be notified when a device is added or removed. |
| USBH_BULK_RemoveNotification() | Removes a callback registered through USBH_BULK_AddNotification. |
| USBH_BULK_Open() | Opens a device given by an index. |
| USBH_BULK_Close() | Closes a handle to an opened device. |
| USBH_BULK_AllowShortRead() | Enables or disables short read mode. |
| USBH_BULK_GetDeviceInfo() | Retrieves information about the BULK device. |
| USBH_BULK_Read() | Reads from the BULK device. |
| USBH_BULK_Write() | Writes data to the BULK device. |
| USBH_BULK_ReadAsync() | Triggers a read transfer to the BULK device. |
| USBH_BULK_WriteAsync() | Triggers a write transfer to the BULK device. |
| USBH_BULK_Cancel() | Cancels a running transfer. |
| USBH_BULK_GetNumBytesInBuffer() | Gets the number of bytes in the receive buffer. |
| USBH_BULK_SetupRequest() | Sends a specific request (class vendor etc) to the device. |

## 12.2.1 USBH_BULK_Init()

### Description

Initializes and registers the BULK device module with emUSB-Host.

### Prototype

```
USBH_STATUS USBH_BULK_Init(const USBH_INTERFACE_MASK * pInterfaceMask);
```

### Parameters

| Parameter | Description |
|---|---|
| pInterfaceMask | Deprecated parameter. Please use USBH_BULK_AddNotification to add new interfaces masks. To be backward compatible the mask added through this parameter will be automatically added when USBH_BULK_RegisterNotification is called. |

### Return value

| | |
|---|---|
| USBH_STATUS_SUCCESS | Success or module already initialized. |

### Additional information

This function can be called multiple times, but only the first call initializes the module. Any further calls only increase the initialization counter. This is useful for cases where the module is initialized from different places which do not interact with each other, To de-initialize the module USBH_BULK_Exit has to be called the same number of times as this function was called.

# 12.2.2 USBH_BULK_Exit()

### Description

Unregisters and de-initializes the BULK device module from emUSB-Host.

### Prototype

```
void USBH_BULK_Exit(void);
```

### Additional information

Has to be called the same number of times `USBH_BULK_Init` was called in order to de-initialize the module. 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_BULK_Exit()`. The only exception is `USBH_BULK_Init()`, which would in turn re-init the module and allow further calls.

# 12.2.3 USBH_BULK_RegisterNotification()

### Description

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

### Prototype

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

### Parameters

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

### Additional information

This function is deprecated, please use function `USBH_BULK_AddNotification()`.

# 12.2.4 USBH_BULK_AddNotification()

### Description

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

### Prototype

```
USBH_STATUS USBH_BULK_AddNotification
                                   (      USBH_NOTIFICATION_HOOK * pHook,
                                          USBH_NOTIFICATION_FUNC * pfNotification,
                                          void                   * pContext,
                                    const USBH_INTERFACE_MASK    * pInterfaceMask);
```

### Parameters

| Parameter | Description |
|---|---|
| pHook | Pointer to a user provided USBH_NOTIFICATION_HOOK variable. |
| pfNotification | Pointer to a function the stack should call when a device is connected or disconnected. |
| pContext | Pointer to a user context that is passed to the callback function. |
| pInterfaceMask | Pointer to a structure of type USBH_INTERFACE_MASK. NULL means that all interfaces will be forwarded to the callback. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Example

```
static USBH_NOTIFICATION_HOOK _Hook;

/*********************************************************************
*
*       _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) {
  (void)pContext;
  switch (Event) {
  case USBH_DEVICE_EVENT_ADD:
    USBH_Logf_Application("**** Device added\n");
    _DevIndex = DevIndex;
    _DevIsReady = 1;
    break;
  case USBH_DEVICE_EVENT_REMOVE:
    USBH_Logf_Application("**** Device removed\n");
    _DevIsReady = 0;
    _DevIndex   = -1;
    break;
  default:;   // Should never happen
  }
}

<...>
USBH_BULK_Init();
USBH_BULK_AddNotification(&_Hook, _cbOnAddRemoveDevice, NULL);
```

```
<...>
```

# 12.2.5 USBH_BULK_RegisterNotification()

### Description

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

### Prototype

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

### Parameters

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

### Additional information

This function is deprecated, please use function USBH_BULK_AddNotification().

# 12.2.6 USBH_BULK_Open()

### Description

Opens a device given by an index.

### Prototype

```
USBH_BULK_HANDLE USBH_BULK_Open(unsigned Index);
```

### Parameters

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

### Return value

| | |
|---|---|
| ≠ 0 | Handle to a BULK device |
| = 0 | Device not available or removed. |

### Additional information

The index of a new connected device is provided to the callback function registered with `USBH_BULK_AddNotification()`.

## 12.2.7 USBH_BULK_Close()

### Description

Closes a handle to an opened device.

### Prototype

```
USBH_STATUS USBH_BULK_Close(USBH_BULK_HANDLE hDevice);
```

### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 12.2.8 USBH_BULK_AllowShortRead()

### Description

Enables or disables short read mode. If enabled, the function USBH_BULK_Read() 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.

### Prototype

```
USBH_STATUS USBH_BULK_AllowShortRead(USBH_BULK_HANDLE hDevice,
                                     U8               AllowShortRead);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| AllowShortRead | Define whether short read mode shall be used or not.<br>• 1 - Allow short read.<br>• 0 - Short read mode disabled. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 12.2.9 USBH_BULK_GetDeviceInfo()

### Description

Retrieves information about the BULK device.

### Prototype

```
USBH_STATUS USBH_BULK_GetDeviceInfo(USBH_BULK_HANDLE        hDevice,
                                    USBH_BULK_DEVICE_INFO * pDevInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| pDevInfo | Pointer to a USBH_BULK_DEVICE_INFO structure that receives the information. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

## 12.2.10 USBH_BULK_Read()

### Description

Reads from the BULK device. Depending of the ShortRead mode (see USBH_BULK_AllowShortRead()), this function will either return as soon as data are available or all data have been read from the device. This function will also return when a set timeout is expired, whatever comes first.

The USB stack can only read complete packets from the USB device. If the size of a received packet exceeds NumBytes than all data that does not fit into the callers buffer (pData) is stored in an internal buffer and will be returned by the next call to USBH_BULK_Read(). See also USBH_BULK_GetNumBytesInBuffer().

To read a null packet, set pData = NULL and NumBytes = 0.

### Prototype

```
USBH_STATUS USBH_BULK_Read(USBH_BULK_HANDLE   hDevice,
                           U8                 EPAddr,
                           U8               * pData,
                           U32                NumBytes,
                           U32              * pNumBytesRead,
                           U32                Timeout);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| EPAddr | Endpoint address. Must be an IN endpoint. |
| pData | Pointer to a buffer to store the read data. |
| NumBytes | Number of bytes to be read from the device. |
| pNumBytesRead | Pointer to a variable which receives the number of bytes read from the device. Can be NULL. |
| Timeout | Timeout in ms. 0 means infinite timeout. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

If the function returns an error code (including USBH_STATUS_TIMEOUT) it already may have read part of the data. The number of bytes read successfully is always stored in the variable pointed to by pNumBytesRead.

# 12.2.11 USBH_BULK_Write()

### Description

Writes data to the BULK device. The function blocks until all data has been written or until the timeout has been reached.

### Prototype

```
USBH_STATUS USBH_BULK_Write(      USBH_BULK_HANDLE   hDevice,
                                  U8                 EPAddr,
                            const U8               * pData,
                                  U32                NumBytes,
                                  U32              * pNumBytesWritten,
                                  U32                Timeout);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| EPAddr | Endpoint address. Must be an OUT endpoint. |
| pData | Pointer to data to be sent. |
| NumBytes | Number of bytes to send. |
| pNumBytesWritten | Pointer to a variable which receives the number of bytes written to the device. Can be NULL. |
| Timeout | Timeout in ms. 0 means infinite timeout. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

If the function returns an error code (including USBH_STATUS_TIMEOUT) it already may have written part of the data. The number of bytes written successfully is always stored in the variable pointed to by pNumBytesWritten.

# 12.2.12 USBH_BULK_ReadAsync()

### Description

Triggers a read transfer to the BULK device. The result of the transfer is received through the user callback. This function will return immediately while the read transfer is done asynchronously.

### Prototype

```
USBH_STATUS USBH_BULK_ReadAsync(USBH_BULK_HANDLE             hDevice,
                                U8                           EPAddr,
                                void                       * pBuffer,
                                U32                          BufferSize,
                                USBH_BULK_ON_COMPLETE_FUNC * pfOnComplete,
                                USBH_BULK_RW_CONTEXT       * pRWContext);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| EPAddr | Endpoint address. Must be an IN endpoint. |
| pBuffer | Pointer to the buffer that receives the data from the device. |
| BufferSize | Size of the buffer in bytes. Must be a multiple of of the maximum packet size of the USB device. |
| pfOnComplete | Pointer to a user function of type USBH_BULK_ON_COMPLETE_FUNC which will be called after the transfer has been completed. |
| pRWContext | Pointer to a USBH_BULK_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function. The member 'pUserContext' may be set before calling USBH_BULK_ReadAsync(). Other members need not be initialized and are set by the function USBH_BULK_ReadAsync(). The memory used for this structure must be valid, until the transaction is completed. |

### Return value

| | |
|---|---|
| = USBH_STATUS_PENDING | Success, the data transfer is queued, the user callback will be called after the transfer is finished. |
| ≠ USBH_STATUS_PENDING | An error occurred, the transfer is not started and user callback will not be called. |

### Example

```
static USBH_BULK_RW_CONTEXT _ReadWriteContext;

<...>

/*********************************************************************
*
*       _OnReadComplete
*/
static void _OnReadComplete(USBH_BULK_RW_CONTEXT * pRWContext) {
  if (pRWContext->Status == USBH_STATUS_SUCCESS) {
    printf("Successfully read %u bytes \n",
           (unsigned int)pRWContext->NumBytesTransferred);
  } else {
    printf("ReadAsync callback returned %s \n",
           USBH_GetStatusStr(pRWContext->Status));
```

```
    // Error handling
  }
  <...>
}

<...>

Status = USBH_BULK_ReadAsync(_hDevice,
                             EPAddr,
                             _acBuffer,
                             NumBytes,
                             _OnReadComplete,
                             &_ReadWriteContext);
if (Status != USBH_STATUS_PENDING) {
  // Error handling.
}
<...>
```

## 12.2.13 USBH_BULK_WriteAsync()

### Description

Triggers a write transfer to the BULK device. The result of the transfer is received through the user callback. This function will return immediately while the write transfer is done asynchronously.

### Prototype

```
USBH_STATUS USBH_BULK_WriteAsync(USBH_BULK_HANDLE             hDevice,
                                 U8                           EPAddr,
                                 void                       * pBuffer,
                                 U32                          BufferSize,
                                 USBH_BULK_ON_COMPLETE_FUNC * pfOnComplete,
                                 USBH_BULK_RW_CONTEXT       * pRWContext);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| EPAddr | Endpoint address. Must be an OUT endpoint. |
| pBuffer | Pointer to a buffer which holds the data. |
| BufferSize | Number of bytes to write. |
| pfOnComplete | Pointer to a user function of type USBH_BULK_ON_COMPLETE_FUNC which will be called after the transfer has been completed. |
| pRWContext | Pointer to a USBH_BULK_RW_CONTEXT structure which will be filled with data after the transfer has been completed and passed as a parameter to the pfOnComplete function. pfOnComplete function. The member 'pUserContext' may be set before calling USBH_BULK_WriteAsync(). Other members need not be initialized and are set by the function USBH_BULK_WriteAsync(). The memory used for this structure must be valid, until the transaction is completed. |

### Return value

| | |
|---|---|
| = USBH_STATUS_PENDING | Success, the data transfer is queued, the user callback will be called after the transfer is finished. |
| ≠ USBH_STATUS_PENDING | An error occurred, the transfer is not started and user callback will not be called. |

### Example

```
static USBH_BULK_RW_CONTEXT _ReadWriteContext;

<...>

/*********************************************************************
*
*       _OnWriteComplete
*/
static void _OnWriteComplete(USBH_BULK_RW_CONTEXT * pRWContext) {
  if (pRWContext->Status == USBH_STATUS_SUCCESS) {
    printf("Successfully written data to the device \n");
  } else {
    printf("WriteAsync callback returned %s \n",
           USBH_GetStatusStr(pRWContext->Status));
    // Error handling
  }
```

```
  <...>
}

<...>


Status = USBH_BULK_WriteAsync(_hDevice,
                              EPAddr,
                              _acBuffer,
                              NumBytes,
                              _OnWriteComplete,
                              &_ReadWriteContext);
if (Status != USBH_STATUS_PENDING) {
  // Error handling.
}
<...>
```

## 12.2.14 USBH_BULK_Cancel()

### Description

Cancels a running transfer.

### Prototype

```
USBH_STATUS USBH_BULK_Cancel(USBH_BULK_HANDLE hDevice,
                             U8               EPAddr);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| EPAddr | Endpoint address. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

This function can be used to cancel a transfer which was initiated by USBH_BULK_ReadAsync()/USBH_BULK_WriteAsync() or USBH_BULK_Read()/USBH_BULK_Write(). In the later case this function has to be called from a different task.

## 12.2.15 USBH_BULK_GetNumBytesInBuffer()

### Description

Gets the number of bytes in the receive buffer.

The USB stack can only read complete packets from the USB device. If the size of a received packet exceeds the number of bytes requested with `USBH_BULK_Read()`, than all data that is not returned by `USBH_BULK_Read()` is stored in an internal buffer.

The number of bytes returned by `USBH_BULK_GetNumBytesInBuffer()` can be read using `USBH_BULK_Read()` out of the buffer without a USB transaction to the USB device being executed.

### Prototype

```
USBH_STATUS USBH_BULK_GetNumBytesInBuffer(USBH_BULK_HANDLE   hDevice,
                                          U8                 EPAddr,
                                          U32              * pRxBytes);
```

### Parameters

| Parameter | Description |
|---|---|
| hDevice | Handle to an open device returned by `USBH_BULK_Open()`. |
| EPAddr | Endpoint address. |
| pRxBytes | Pointer to a variable which receives the number of bytes in the receive buffer. |

### Return value

`USBH_STATUS_SUCCESS` on success or error code on failure.

### Example

```
//
// Read only ONE byte to trigger the read transfer.
// This means that the remaining bytes are in the internal packet buffer!
//
USBH_BULK_Read(hDevice, EPAddr, acData, 1, &NumBytes, Timeout);
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_BULK_GetNumBytesInBuffer we can check how many bytes are still in the
  // internal buffer (if any) and read those as well.
  //
  if (USBH_BULK_GetNumBytesInBuffer(hDevice, EPAddr, &RxBytes) == USBH_STATUS_SUCCESS) {
    //
    // Read the remaining bytes.
    //
    if (RxBytes > 0) {
      USBH_BULK_Read(hDevice, EPAddr, &acData[1], RxBytes, &NumBytes, Timeout);
    }
  }
}
```

# 12.2.16 USBH_BULK_SetupRequest()

### Description

Sends a specific request (class vendor etc) to the device.

### Prototype

```
USBH_STATUS USBH_BULK_SetupRequest(USBH_BULK_HANDLE   hDevice,
                                   U8                 RequestType,
                                   U8                 Request,
                                   U16                wValue,
                                   U16                wIndex,
                                   void             * pData,
                                   U32              * pNumBytesData,
                                   U32                Timeout);
```

### Parameters

| Parameter | Description |
| --- | --- |
| hDevice | Handle to an open device returned by USBH_BULK_Open(). |
| RequestType | IN/OUT direction. |
| Request | Request code in the setup request. |
| wValue | wValue in the setup request. |
| wIndex | wIndex in the setup request. |
| pData | Additional data for the setup request. |
| pNumBytesData | Number of data to be received/sent in pData. |
| Timeout | Timeout in ms. 0 means infinite timeout. |

### Return value

USBH_STATUS_SUCCESS on success or error code on failure.

### Additional information

wLength which is normally part of the setup packet will be determined given by the pNum-Bytes and pData. In case no pBuffer is given, wLength will be 0.

## 12.3 Data structures

This chapter describes the emUSB-Host BULK driver data structures.

| Structure | Description |
| --- | --- |
| USBH_BULK_DEVICE_INFO | Structure containing information about a BULK device. |
| USBH_BULK_EP_INFO | Structure containing information about an endpoint. |
| USBH_BULK_RW_CONTEXT | Contains information about a completed, asynchronous transfers. |

## 12.3.1 USBH_BULK_DEVICE_INFO

### Description

Structure containing information about a BULK device.

### Type definition

```
typedef struct {
  U16                VendorId;
  U16                ProductId;
  U8                 Class;
  U8                 SubClass;
  U8                 Protocol;
  U8                 AlternateSetting;
  USBH_SPEED         Speed;
  U8                 InterfaceNo;
  U8                 NumEPs;
  USBH_BULK_EP_INFO  EndpointInfo[];
  USBH_DEVICE_ID     DeviceId;
  USBH_INTERFACE_ID  InterfaceID;
} USBH_BULK_DEVICE_INFO;
```

### Structure members

| Member | Description |
|---|---|
| VendorId | The Vendor ID of the device. |
| ProductId | The Product ID of the device. |
| Class | The interface class. |
| SubClass | The interface sub class. |
| Protocol | The interface protocol. |
| AlternateSetting | The current alternate setting |
| Speed | The USB speed of the device, see USBH_SPEED. |
| InterfaceNo | Index of the interface (from USB descriptor). |
| NumEPs | Number of endpoints. |
| EndpointInfo | Information about all endpoints. |
| DeviceId | 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. |
| InterfaceID | Interface ID of the device. |

## 12.3.2 USBH_BULK_EP_INFO

### Description

Structure containing information about an endpoint.

### Type definition

```
typedef struct {
  U8   Addr;
  U8   Type;
  U8   Direction;
  U16  MaxPacketSize;
} USBH_BULK_EP_INFO;
```

### Structure members

| Member | Description |
|---|---|
| Addr | Endpoint Address. |
| Type | Endpoint Type (see USB_EP_TYPE_... macros). |
| Direction | Endpoint direction (see USB_..._DIRECTION macros). |
| MaxPacketSize | Maximum packet size for the endpoint. |

## 12.3.3 USBH_BULK_RW_CONTEXT

### Description

Contains information about a completed, asynchronous transfers. Is passed to the USBH_BULK_ON_COMPLETE_FUNC user callback when using asynchronous write and read. When this structure is passed to USBH_BULK_ReadAsync() or USBH_BULK_WriteAsync() its member need not to be initialized.

### Type definition

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

### Structure members

| Member | Description |
|---|---|
| pUserContext | Pointer to a user context. Can be arbitrarily used by the application. |
| Status | Result status of the asynchronous transfer. |
| NumBytesTransferred | Number of bytes transferred. |
| pUserBuffer | Pointer to the buffer provided to USBH_BULK_ReadAsync() or USBH_BULK_WriteAsync(). |
| UserBufferSize | Size of the buffer as provided to USBH_BULK_ReadAsync() or USBH_BULK_WriteAsync(). |

# 12.4 Type definitions

This chapter describes the types defined in the header file `USBH_BULK.h`.

| Type | Description |
|---|---|
| USBH_BULK_ON_COMPLETE_FUNC | Function called on completion of an asynchronous transfer. |

## 12.4.1 USBH_BULK_ON_COMPLETE_FUNC

### Description

Function called on completion of an asynchronous transfer. Used by the functions USB-H_BULK_ReadAsync() and USBH_BULK_WriteAsync().

### Type definition

```
typedef void USBH_BULK_ON_COMPLETE_FUNC(USBH_BULK_RW_CONTEXT * pRWContext);
```

### Parameters

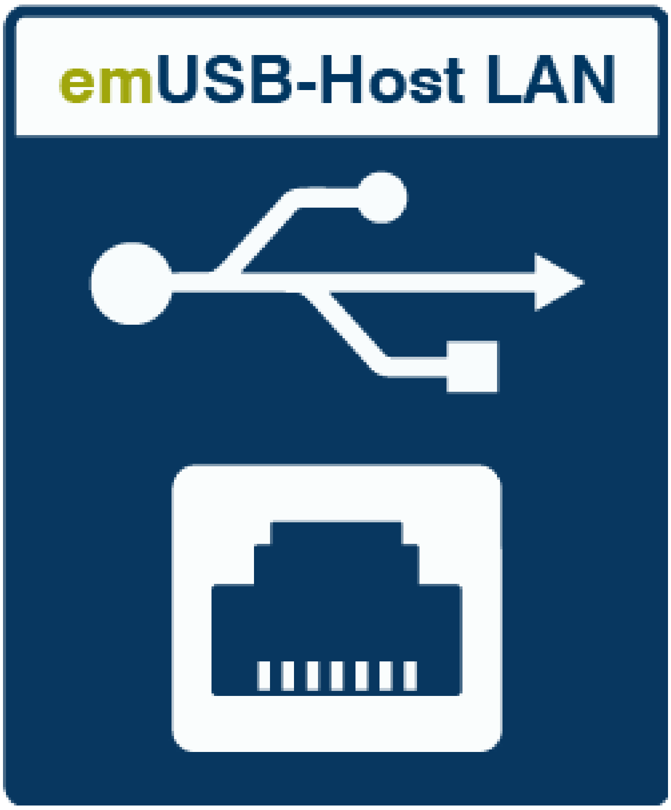
| Parameter | Description |
|---|---|
| pRWContext | Pointer to a USBH_BULK_RW_CONTEXT structure. |

# Chapter 13

# LAN component (Add-On)

This chapter describes the optional emUSB-Host add-on "LAN". It allows interfacing Ethernet-over-USB adapters with embOS/IP.



# 13.1 Introduction

The LAN software component of emUSB-Host allows communication with Ethernet-over-USB adapters. These devices usually implement the CDC-ECM, RNDIS protocol or a proprietary protocol from the company ASIX. All above protocols allow the transfer of Ethernet packets over USB. emUSB-Host LAN provides a seamless interface with embOS/IP irrespective of the underlying USB protocol thereby allowing devices without Ethernet connectors to connect with a network.

This chapter provides an explanation of the LAN software component functions available to application developers. All the functions and data types of this add-on are prefixed with 'USBH_LAN_'.

## 13.1.1 Overview

embOS/IP adds Ethernet interfaces for as many Ethernet-over-USB adapters as are expected to be used with the product. The interfaces are initially "down". emUSB-Host LAN accommodates multiple underlying classes to support different adapters. Each registered LAN driver notifies the main LAN module when a device matching the LAN driver's supported class (ASIX, RNDIS or CDC-ECM) has enumerated. The LAN module in turn notifies the IP stack that an interface is "up" and communication begins. emUSB-Host LAN is supported with version 3.30 of embOS/IP and higher.

## 13.1.2 Features

The following features are provided:

- Compatibility with different Ethernet-over-USB adapters.
- Integration with embOS/IP

## 13.1.3 Example code

Any embOS/IP example can be used.

## 13.2 IP_Config_USBH_LAN.c in detail

The embOS/IP configuration file IP_Config_USBH_LAN.c is a sample configuration for using emUSB-Host LAN as an interface with embOS/IP.

The function IP_X_Config is the main configuration function of the embOS/IP stack. In this sample the NUM_INSTANCES define (4 by default) is used to determine how many interfaces are registered. Each embOS/IP interface corresponds to one Ethernet-over-USB adapter on the emUSB-Host LAN side. This means that in the default configuration 4 adapters can be used simultaneously (e.g. 4x CDC-ECM adapter or 2x ASIX, 1x CDC-ECM and 1x RNDIS or any other combination of the supported protocols).

```
int aIFaceId[NUM_INSTANCES];
<...>
IP_ConfigMaxIFaces(NUM_INSTANCES);
for (i = 0; i < NUM_INSTANCES; i++) {
  aIFaceId[i] = IP_AddEtherInterface(&IP_Driver_USBH);
  mtu = 1500;
  IP_SetMTU(aIFaceId[i], mtu);
  IP_DHCPC_Activate(aIFaceId[i], "USBH_LAN", NULL, NULL);
<...>
}
```

The sample configuration initializes emUSB-Host and the emUSB-Host LAN component in the same function. This is for convenience only, you can initialize emUSB-Host anywhere inside your application. The initialization starts the stack and the LAN module allowing the Ethernet-over-USB adapters to enumerate.

```
<...>
USBH_Init();
OS_CREATETASK(&_TCBMain, "USBH_Task", USBH_Task, TASK_PRIO_USBH_MAIN, _StackMain);
OS_CREATETASK(&_TCBIsr, "USBH_isr", USBH_ISRTask, TASK_PRIO_USBH_ISR, _StackIsr);
USBH_LAN_Init();
USBH_LAN_RegisterDriver(&USBH_LAN_DRIVER_ASIX);
USBH_LAN_RegisterDriver(&USBH_LAN_DRIVER_ECM);
USBH_LAN_RegisterDriver(&USBH_LAN_DRIVER_RNDIS);
<...>
```

# 13.3 API Functions

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

| Function | Description |
|---|---|
| USBH_LAN_Init() | Initializes and registers the LAN component with emUSB-Host. |
| USBH_LAN_RegisterDriver() | Registers a device specific driver (CDC-ECM, ASIX, RNDIS) with the LAN component. |
| USBH_LAN_Exit() | De-initializes the LAN component. |

## 13.3.1 USBH_LAN_Init()

### Description

Initializes and registers the LAN component with emUSB-Host.

### Prototype

```
USBH_STATUS USBH_LAN_Init(void);
```

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Success |
| ≠ USBH_STATUS_SUCCESS | Could not initialize LAN component. |

# 13.3.2 USBH_LAN_RegisterDriver()

### Description

Registers a device specific driver (CDC-ECM, ASIX, RNDIS) with the LAN component.

### Prototype

```
USBH_STATUS USBH_LAN_RegisterDriver(const USBH_LAN_DRIVER * pDriver);
```

### Parameters

| Parameter | Description |
|---|---|
| pDriver | Pointer to an LAN driver structure of type USBH_LAN_DRIVER. Currently the following drivers are available:<br>• USBH_LAN_DRIVER_ASIX<br>• USBH_LAN_DRIVER_ECM<br>• USBH_LAN_DRIVER_RNDIS |

### Return value

| | |
|---|---|
| = USBH_STATUS_SUCCESS | Success |
| ≠ USBH_STATUS_SUCCESS | Could not register LAN driver. |

## 13.3.3 USBH_LAN_Exit()

### Description

De-initializes the LAN component.

### Prototype

```
void USBH_LAN_Exit(void);
```

# Chapter 14

# 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.

# 14.1 Introduction

## 14.1.1 Overview

USB On-The-Go (OTG) allows two USB devices to communicate with 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.

## 14.1.2 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.

## 14.1.3 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(25);
      BSP_ToggleLED(0);
      USB_OTG_OS_Delay(25);
      BSP_ToggleLED(1);
    }
```

```
    //
    // 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();
    }
  }
}
```

# 14.2 OTG Driver

## 14.2.1 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.

# 14.3 API Functions

This chapter describes the emUSB-OTG API functions.

| Function | Description |
|---|---|
| `USB_OTG_Init()` | Initializes the core. |
| `USB_OTG_DeInit()` | Deinitialize the complete OTG module. |
| `USB_OTG_GetIdState()` | Returns the current state of the USB OTG ID pin. |
| `USB_OTG_GetVBUSState()` | Returns the current state of the VBUS via an OTG transceiver. |
| `USB_OTG_IsSessionValid()` | Returns whether the OTG transceiver has marked the session as valid. |
| `USB_OTG_AddDriver()` | Adds a OTG driver to the OTG stack. |
| `USB_OTG_X_Config()` | User-provided function which configures the emUSB-OTG stack. |

#### 14.3.0.1 USB_OTG_Init()

**Description**

Initializes the core. It initially initializes the OS-Layer, calls the driver initialization callback.

**Prototype**

```
void USB_OTG_Init(void);
```

## 14.3.0.2 USB_OTG_DeInit()

### Description

Deinitialize the complete OTG module. It removes/releases all OS-layer relevant resources and calls the driver deinitialization callback.

### Prototype

```
void USB_OTG_DeInit(void);
```

## 14.3.0.3 USB_OTG_GetIdState()

### Description

Returns the current state of the USB OTG 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 |

# 14.3.0.4 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 millivolt.

### 14.3.0.5 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. |

### 14.3.0.6 USB_OTG_AddDriver()

#### Description

Adds a OTG driver to the OTG stack. This function is generally called in the USB_OTG_X_AddDriver.

#### Prototype

```
void USB_OTG_AddDriver(const USB_OTG_HW_DRIVER * pDriver);
```

#### Parameters

| Parameter | Description |
|---|---|
| pDriver | Pointer to the driver structure. |

#### 14.3.0.7 USB_OTG_X_Config()

**Description**

User provided function which configures the USB OTG stack.

**Prototype**

void USB_OTG_X_Config(void);

**Additional information**

This function is called by the start-up code of the USB OTG stack from USB_OTG_Init(). This function should initialize all necessary clocks and pins required for the OTG operation of your controller.

**Example**

```
/*********************************************************************
*
*       USB_OTG_X_Config
*/
void USB_OTG_X_Config(void) {
  _InitClocks();
  _InitPins();
  USB_OTG_AddDriver(&USB_OTG_Driver_ST_STM32F2xxFS);
}
```

# Chapter 15

# Configuring emUSB-Host

This chapter explains how to configure emUSB-Host.

# 15.1 Runtime configuration

The configuration of emUSB-Host for a target hardware is done at runtime: The emUSB-Host stack calls a function named `USBH_X_Config`, that must be provided by the application. This function performs board specific hardware initialization like configuring I/O pins of the MCU, setting up PLL and clock divider necessary for USB and installing the interrupt service routine for USB.

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 before the host controller driver Add-function is called.

Additionally all runtime configuration of the USB stack is done in this function, for example:

- Assign memory to be used by the emUSB-Host stack.
- Select an appropriate driver for the USB host controller.
- Set driver specific parameters like base address of the controller of transfer buffer sizes.
- Set debug message output filter.
- Set a memory address translation routine (if a MMU is used).
- Enable HUB support.

Sample configurations for popular evaluation boards are supplied with the driver shipment. They can be found in files called `USBH_Config_<TargetName>.c` in the folders `BSP/<BoardName>/Setup`. This files can be used as a template for a customized configuration.

## 15.1.1 USBH_X_Config()

### Description

Initialize USB hardware and configure the USB-Host stack. This function is called by the startup code of the emUSB-Host stack from `USBH_Init()`. This is the place where a hardware driver can be added and configured.

### Prototype

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

### Example

```
void USBH_X_Config(void) {
  //
  // Assigning memory should be the first thing
  //
  USBH_AssignMemory(&_aPool[0], ALLOC_SIZE);
  USBH_AssignTransferMemory(&_aTransferBufferPool[0], ALLOC_TRANSFER_SIZE);
  //
  // Allow external hubs
  //
  USBH_ConfigSupportExternalHubs(1);
  //
  // Wait 300ms for a new connected device
  //
  USBH_ConfigPowerOnGoodTime(300);
  //
  // Define log and warn filter
  //
  USBH_SetWarnFilter(0xFFFFFFFF);             // Output all warnings.
```

```
  USBH_SetLogFilter(0
                    | USBH_MTYPE_INIT
                    | USBH_MTYPE_APPLICATION
                    );
  //
  // Initialize USB hardware
  //
  _InitUSBHw();
  //
  // Add EHCI driver
  //
  USBH_EHCI_EX_Add((void*)(USB_EHCI_BASE_ADDR + 0x100));
  //
  // Install interrupt service routine
  //
  BSP_USBH_InstallISR_Ex(USB0_IRQn, _ISR, USB_ISR_PRIO);
}
```

## Configuration functions

Functions that may or must be used in USBH_X_Config are listed in the following table. Additional driver dependant functions exist for every USB host controller driver, see *Host controller specifics* on page 375.

| Function | Description |
|---|---|
| USBH_AssignMemory() | Assigns a memory area that will be used by the memory management functions for allocating memory. |
| USBH_AssignTransferMemory() | Assigns a memory area for a heap that will be used for allocating DMA memory. |
| USBH_Config_SetV2PHandler() | Sets a virtual address to physical address translator. |
| USBH_ConfigPowerOnGoodTime() | Configures the default power on time that the host waits after connecting a device before starting to communicate with the device. |
| USBH_ConfigSupportExternalHubs() | Enable support for external USB hubs. |
| USBH_ConfigTransferBufferSize() | Configures the size of a copy buffer that can be used if the USB controller has limited access to the system memory. |
| USBH_SetCacheConfig() | Configures cache related functionality that might be required by the stack for several purposes such as cache handling in drivers. |
| USBH_SetOnSetPortPower() | Sets a callback for the set-port-power driver function. |
| USBH_SetLogFilter() | Sets a mask that defines which logging message should be logged. |
| USBH_SetWarnFilter() | Adds an additional filter condition to the mask which specifies the warning messages that should be displayed. |

# 15.2 Compile-time configuration

emUSB-Host can be used without changing any of the compile-time flags. All compile-time configuration flags are preconfigured with valid values which match the requirements of most applications.

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.

The following types of configuration macros exist:

**Numerical values "N"**

Numerical values are used somewhere in the code in place of a numerical constant.

**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).

| Type | Symbolic name | Default | Description |
|---|---|---|---|
| N | USBH_DEBUG | 0 | emUSB-Host can be configured to display debug information at higher debug levels to locate an error or potential problems. 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. |
| N | USBH_RESET_RETRY_COUNTER | 5 | If an error is encountered during USB enumeration the process is repeated USBH_RESET_RETRY_COUNTER times before the port is finally disabled. |
| N | USBH_DELAY_FOR_REENUM | 1000 | Delay time in ms before a new enumeration of a USB device is retried after an error. |
| 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. |
| 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. |

# 15.3 Host controller specifics

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

This section lists the drivers which require special configuration and describes how to configure those drivers.

## 15.3.1 EHCI driver

Normally EHCI controllers only handle high-speed USB devices. Some EHCI controllers contain a transaction translator (TT) that enables them to handle full- and low-speed devices also. There are different Add-functions to configure the driver for host controllers with or without a TT.

### Systems with cached memory

If the EHCI driver is installed on a system using cached (data) memory, the following requirements must be considered:

- A special region of RAM is necessary that can be accessed non-cached and non-buffered by the CPU. The USB host controller must also be able to access this area via DMA.
- The non-cached RAM area must be provided to the USB stack using the function `USBH_AssignTransferMemory()`. The memory area must be cache clean before calling the function `USBH_AssignTransferMemory()`.
- If the physical address is not equal to the virtual address of the non-cached memory area (address translation by an MMU), a mapping function must be installed using `USBH_Config_SetV2PHandler()`. The translated addresses (physical addresses) are used for DMA by the host controller.
- Cache functions that may be set with `USBH_SetCacheConfig()` are **not** used by the driver.
- The function `USBH_EHCI_Config_UseTransferBuffer()` must be called, in order to tell the driver that application defined buffers can not be accessed directly via DMA.

### Systems without cached memory

On systems without cache there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

### 15.3.1.1 EHCI driver specific configuration functions

| Function | Description |
|---|---|
| `USBH_EHCI_Add()` | Adds a HS capable EHCI controller to the stack. |
| `USBH_EHCI_EX_Add()` | Adds a LS/FS/HS capable EHCI controller to the stack. |
| `USBH_RZG1E_Add()` | Adds a HS capable EHCI controller to the stack. |
| `USBH_RT1050_Add()` | Adds a HS capable EHCI controller to the stack. |
| `USBH_EHCI_Config_SetM2MEndianMode()` | Setups the internal EHCI memory to memory transfer endianess mode. |
| `USBH_EHCI_Config_UseTransferBuffer()` | Configures the driver to use temporary transfer buffers instead using the user buffer directly. |

### 15.3.1.2 USBH_EHCI_Add()

#### Description

Adds a HS capable EHCI controller to the stack.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the EHCI controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

## 15.3.1.3 USBH_EHCI_EX_Add()

### Description

Adds a LS/FS/HS capable EHCI controller to the stack.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the EHCI controllers register set. |

### Return value

Reference to the added host controller (0-based index).

### 15.3.1.4 USBH_RZG1E_Add()

#### Description

Adds a HS capable EHCI controller to the stack. This EHCI initialisation function is specific to the RZG1E series.

#### Prototype

```
U32 USBH_RZG1E_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the EHCI controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.1.5 USBH_RT1050_Add()

#### Description

Adds a HS capable EHCI controller to the stack. This EHIC initialisation function is specific to the RT1050 series.

#### Prototype

```
U32 USBH_RT1050_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
| --- | --- |
| pBase | Pointer to the base of the EHCI controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

#### Additional information

Suspend and resume of USB devices is not supported for the RT1050 series EHCI controller. Due to a hardware issue devices connected to the roothub port must stay connected for at least 250ms before they are removed. Should a device be removed before that the USB controller will crash any only recover after a power cycle of the MCU.

### 15.3.1.6 USBH_EHCI_Config_SetM2MEndianMode()

#### Description

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

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller returned by USBH_EHCI_EX_Add(). |
| Endian | • USBH_EHCI_M2M_ENDIAN_MODE_LITTLE - use little endian mode for memory-2-memory (DMA) transfers<br>• USBH_EHCI_M2M_ENDIAN_MODE_BIG - use big endian mode for memory-2-memory (DMA) transfers |

## 15.3.1.7 USBH_EHCI_Config_UseTransferBuffer()

### Description

Configures the driver to use temporary transfer buffers instead using the user buffer directly. This is necessary if the MCU uses cache.

### Prototype

```
void USBH_EHCI_Config_UseTransferBuffer(U32 HCIndex);
```

### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller, returned by USBH_EHCI_Add() or USBH_EHCI_EX_Add(). |

## 15.3.2 Synopsys DWC2 driver

Using this driver there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

If the Synopsys driver operates in high-speed mode and is installed on a system using cached (data) memory, cache functions for cleaning and invalidating cache lines must be provided and set with `USBH_SetCacheConfig()`.

On some MCUs the USB controller is not able to access all RAM areas the application uses. In this case a function can be installed that checks for memory valid for DMA access. If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

### 15.3.2.1 Restrictions

#### Low speed devices

On STM32Fxxx MCUs low speed USB devices connected via an external hub are not recognized due to a hardware limitation. If connected in this way it may happen, that the host controller gets disturbed and blocked. In order to return to normal operation, a reset of the controller and the external hub may be necessary.

The issue seems to be related to the internal PHY of the STM32F MCUs. It usually not occurs, if the high speed USB controller is used in connection with an external (high speed) PHY.

#### Split transactions

If the host controller operates in high-speed and a full or low-speed device is connected via an external hub, the driver uses split transactions to access the device. Split transactions will only work reliable, if

- The task running `USBH_ISRTask()` must have the highest priority in the system.
- The task running `USBH_Task()` must have the second highest priority.
- All other task must have a lower priority than `USBH_Task()`.
- Both USB tasks must not be delayed by any interrupt service routine for more than 500 µs.

### 15.3.2.2 Synopsys driver specific configuration functions

| Function | Description |
|---|---|
| `USBH_STM32_Add()` | Adds a Synopsys DWC2 full speed controller of a STM32F107 device to the stack. |
| `USBH_STM32F2_FS_Add()` | Adds a Synopsys DWC2 full speed controller of a STM32F2xx or STM32F4xx device to the stack. |
| `USBH_STM32F2_HS_Add()` | Adds a Synopsys DWC2 high speed controller of a STM32F2xx or STM32F4xx device to the stack. |
| `USBH_STM32F2_HS_AddEx()` | Adds a Synopsys DWC2 high speed controller of a STM32F2xx or STM32F4xx device to the stack. |
| `USBH_STM32F2_HS_SetCheckAddress()` | Installs a function that checks if an address can be used for DMA transfers. |
| `USBH_STM32F7_FS_Add()` | Adds a Synopsys DWC2 full speed controller of a STM32F7xx device to the stack. |
| `USBH_STM32F7_HS_Add()` | Adds a Synopsys DWC2 high speed controller of a STM32F7xx device to the stack. |

| Function | Description |
| --- | --- |
| USBH_STM32F7_HS_AddEx() | Adds a Synopsys DWC2 high speed controller of a STM32F7xx or STM32F7xx device to the stack. |
| USBH_STM32F7_HS_SetCheckAddress() | Installs a function that checks if an address can be used for DMA transfers. |
| USBH_STM32H7_HS_Add() | Adds a Synopsys DWC2 high speed controller of a STM32H7xx device to the stack. |
| USBH_STM32H7_HS_AddEx() | Adds a Synopsys DWC2 high speed controller of a STM32H7xx or STM32H7xx device to the stack. |
| USBH_STM32H7_HS_SetCheckAddress() | Installs a function that checks if an address can be used for DMA transfers. |
| USBH_XMC4xxx_FS_Add() | Adds a Synopsys DWC2 full speed controller of a XMC4xxx device to the stack. |

### 15.3.2.3 USBH_STM32_Add()

#### Description

Adds a Synopsys DWC2 full speed controller of a STM32F107 device to the stack.

#### Prototype

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

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.2.4 USBH_STM32F2_FS_Add()

#### Description

Adds a Synopsys DWC2 full speed controller of a STM32F2xx or STM32F4xx device to the stack.

#### Prototype

```
U32 USBH_STM32F2_FS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.2.5 USBH_STM32F2_HS_Add()

#### Description

Adds a Synopsys DWC2 high speed controller of a STM32F2xx or STM32F4xx device to the stack.

#### Prototype

```
U32 USBH_STM32F2_HS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.2.6 USBH_STM32F2_HS_AddEx()

#### Description

Adds a Synopsys DWC2 high speed controller of a STM32F2xx or STM32F4xx device to the stack.

#### Prototype

```
U32 USBH_STM32F2_HS_AddEx(void * pBase,
                          U8     PhyType);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |
| PhyType | • 0 - use external PHY connected via ULPI interface.<br>• 1 - use internal full speed PHY. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.2.7 USBH_STM32F2_HS_SetCheckAddress()

#### Description

Installs a function that checks if an address can be used for DMA transfers. Installed function must return 0, if DMA access is allowed for the given address, 1 otherwise.

#### Prototype

```
void USBH_STM32F2_HS_SetCheckAddress
                                (USBH_CHECK_ADDRESS_FUNC * pfCheckValidDMAAddress);
```

#### Parameters

| Parameter | Description |
| --- | --- |
| pfCheckValidDMAAd-dress | Pointer to the function. |

#### Additional information

If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

## 15.3.2.8 USBH_STM32F7_FS_Add()

### Description

Adds a Synopsys DWC2 full speed controller of a STM32F7xx device to the stack.

### Prototype

```
U32 USBH_STM32F7_FS_Add(void * pBase);
```

### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

### Return value

Reference to the added host controller (0-based index).

### 15.3.2.9 USBH_STM32F7_HS_Add()

#### Description

Adds a Synopsys DWC2 high speed controller of a STM32F7xx device to the stack.

#### Prototype

```
U32 USBH_STM32F7_HS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

# 15.3.2.10 USBH_STM32F7_HS_AddEx()

## Description

Adds a Synopsys DWC2 high speed controller of a STM32F7xx or STM32F7xx device to the stack.

## Prototype

```
U32 USBH_STM32F7_HS_AddEx(void * pBase,
                          U8     PhyType);
```

## Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |
| PhyType | • 0 - use external PHY connected via ULPI interface.<br>• 1 - use internal full speed PHY. |

## Return value

Reference to the added host controller (0-based index).

## 15.3.2.11 USBH_STM32F7_HS_SetCheckAddress()

### Description

Installs a function that checks if an address can be used for DMA transfers. Installed function must return 0, if DMA access is allowed for the given address, 1 otherwise.

### Prototype

```
void USBH_STM32F7_HS_SetCheckAddress
                                (USBH_CHECK_ADDRESS_FUNC * pfCheckValidDMAAddress);
```

### Parameters

| Parameter | Description |
|---|---|
| pfCheckValidDMAAddress | Pointer to the function. |

### Additional information

If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

### 15.3.2.12 USBH_STM32H7_HS_Add()

#### Description

Adds a Synopsys DWC2 high speed controller of a STM32H7xx device to the stack.

#### Prototype

```
U32 USBH_STM32H7_HS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
| --- | --- |
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.2.13 USBH_STM32H7_HS_AddEx()

#### Description

Adds a Synopsys DWC2 high speed controller of a STM32H7xx or STM32H7xx device to the stack.

#### Prototype

```
U32 USBH_STM32H7_HS_AddEx(void * pBase,
                          U8     PhyType);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |
| PhyType | • 0 - use external PHY connected via ULPI interface.<br>• 1 - use internal full speed PHY. |

#### Return value

Reference to the added host controller (0-based index).

## 15.3.2.14 USBH_STM32H7_HS_SetCheckAddress()

### Description

Installs a function that checks if an address can be used for DMA transfers. Installed function must return 0, if DMA access is allowed for the given address, 1 otherwise.

### Prototype

```
void USBH_STM32H7_HS_SetCheckAddress
                              (USBH_CHECK_ADDRESS_FUNC * pfCheckValidDMAAddress);
```

### Parameters

| Parameter | Description |
|---|---|
| pfCheckValidDMAAd-dress | Pointer to the function. |

### Additional information

If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

#### 15.3.2.15 USBH_XMC4xxx_FS_Add()

**Description**

Adds a Synopsys DWC2 full speed controller of a XMC4xxx device to the stack.

**Prototype**

```
U32 USBH_XMC4xxx_FS_Add(void * pBase);
```

**Parameters**

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

**Return value**

Reference to the added host controller (0-based index).

## 15.3.3 OHCI driver

OHCI controllers handle full-speed and low-speed USB devices.

### Systems with cached memory

If the OHCI driver is installed on a system using cached (data) memory, the following requirements must be considered:

- A special region of RAM is necessary that can be accessed non-cached and non-buffered by the CPU. The USB host controller must also be able to access this area via DMA.
- The non-cached RAM area must be provided to the USB stack using the function `USBH_AssignTransferMemory()`. The memory area must be cache clean before calling the function `USBH_AssignTransferMemory()`.
- If the physical address is not equal to the virtual address of the non-cached memory area (address translation by an MMU), a mapping function must be installed using `USBH_Config_SetV2PHandler()`. The translated addresses (physical address) are used for DMA by the host controller.
- Cache functions that may be set with `USBH_SetCacheConfig()` are **not** used by the driver.

### Systems without cached memory

On systems without cache there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

On systems without cache and where the OHCI controller has access to the memory where application buffers are located `USBH_OHCI_Config_UseZeroCopy()` can be used to improve performance.

### 15.3.3.1 OHCI driver specific configuration functions

| Function | Description |
|---|---|
| USBH_OHCI_Add() | Adds a full-speed capable OHCI controller to the stack. |
| USBH_OHCI_Config_UseZeroCopy() | Configures the driver to use the user buffer directly instead of using allocated transfer buffers. |
| USBH_OHCI_LPC546_Add() | Adds a full-speed capable OHCI controller to the stack. |

## 15.3.3.2 USBH_OHCI_Add()

### Description

Adds a full-speed capable OHCI controller to the stack.

### Prototype

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

### Parameters

| Parameter | Description |
| --- | --- |
| pBase | Pointer to the base of the OHCI controllers register set. |

### Return value

Reference to the added host controller (0-based index).

### 15.3.3.3 USBH_OHCI_Config_UseZeroCopy()

#### Description

Configures the driver to use the user buffer directly instead of using allocated transfer buffers. This can be enabled when the MCU does not use cache and when the controller has access to the memories where the user buffers are located.

#### Prototype

```
void USBH_OHCI_Config_UseZeroCopy(U32 HCIndex);
```

#### Parameters

| Parameter | Description |
|---|---|
| HCIndex | Index of the host controller. |

### 15.3.3.4 USBH_OHCI_LPC546_Add()

#### Description

Adds a full-speed capable OHCI controller to the stack. This add function, enables workarounds inside the OHCI driver for the LPC546xx series. One of these workarounds creates a limitation where an interval timeout of 1 can not be guaranteed for interrupt endpoints as each interrupt endpoint transfer completion has to be delayed by up to two frames.

#### Prototype

```
U32 USBH_OHCI_LPC546_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the OHCI controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

## 15.3.4 Kinetis USBOTG FS driver

KinetisFS controllers handle full-speed and low-speed USB devices.

### Systems without cached memory

On systems without cache there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

### Restrictions

- WFI instruction can not be used when the USBOTG module is used. This is a hardware issue, see errata e7166 for chip masks 4N96B, 1N96B, 3N96B.
- When the controller accesses the flash memory (e.g. when an application writes to a device from a const char array) it is necessary to allow the controller access to the flash area and to set the bus master priority to the highest level, otherwise read accesses to the flash may be stalled which results in the controller getting less data than expected and respective follow-up errors.
- This controller can only schedule a single transaction at a time. This means that hubs can not be used with this host controller. Furthermore composite devices will not work properly if the application communicates on multiple interfaces at once. Devices which contain an Interrupt IN endpoint which needs to be constantly polled by the host controller will also have issues as the Interrupt IN endpoint will hog the only communication pipe, a good sample for this is the CDC class which consists of Bulk IN, Bulk OUT and Interrupt IN endpoints, in this case the controller will be constantly busy polling the Interrupt IN endpoint and will not be able to communicate via the Bulk endpoints. To work around the Interrupt IN issue emUSB-Host provides the functions `USBH_CDC_SetConfigFlags()` and `USBH_HID_ConfigureAllowLEDUpdate()`.

### 15.3.4.1 KinetisFS driver specific configuration functions

| Function | Description |
|---|---|
| `USBH_KINETIS_FS_Add()` | Adds a full-speed capable KinetisFS controller to the stack. |

## 15.3.4.2 USBH_KINETIS_FS_Add()

### Description

Adds a full-speed capable KinetisFS controller to the stack.

### Prototype

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

### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the KinetisFS controllers register set. |

### Return value

Reference to the added host controller (0-based index).

## 15.3.5 Renesas driver

Using this driver there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

### 15.3.5.1 Restrictions

The full speed version of the controller can only handle up to 5 devices at once. High speed controller can handle up to 10 devices. External hub support is available but only works under some circumstances. It seems that the concurrent transfers is not possible with higher bandwidth and multiple devices.

#### Low speed devices

RX62x and RX63x series contains a USB controller where low speed device such as mice, keyboards are not recognized properly. Therefore for these device, low-speed support is disabled. You may see in debug builds, that the warning message that this is not possible.

### 15.3.5.2 Renesas driver specific configuration functions

| Function | Description |
|---|---|
| `USBH_RX11_Add()` | Adds a Renesas USB controller of a RX11x device to the stack. |
| `USBH_RX23_Add()` | Adds a Renesas USB controller of a RX23x device to the stack. |
| `USBH_RX62_Add()` | Adds a Renesas USB controller of a RX62x device to the stack. |
| `USBH_RX63_Add()` | Adds a Renesas USB controller of a RX63x device to the stack. |
| `USBH_RX64_Add()` | Adds a Renesas USB controller of a RX64x device to the stack. |
| `USBH_RX65_Add()` | Adds a Renesas USB controller of a RX65x device to the stack. |
| `USBH_RX71_FS_Add()` | Adds a Renesas FS USB controller of a RX71x device to the stack. |
| `USBH_RX71_HS_Add()` | Adds a Renesas HS USB controller of a RX71x device to the stack. |
| `USBH_RZA1_Add()` | Adds a Renesas USB controller of a RZA1 device to the stack. |
| `USBH_Synergy_FS_Add()` | Adds a Renesas FS USB controller of a Synergy device to the stack. |
| `USBH_Synergy_HS_Add()` | Adds a Renesas HS USB controller of a Synergy device to the stack. |

### 15.3.5.3 USBH_RX11_Add()

#### Description

Adds a Renesas USB controller of a RX11x device to the stack.

#### Prototype

```
U32 USBH_RX11_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.5.4 USBH_RX23_Add()

#### Description

Adds a Renesas USB controller of a RX23x device to the stack.

#### Prototype

```
U32 USBH_RX23_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

#### 15.3.5.5 USBH_RX62_Add()

**Description**

Adds a Renesas USB controller of a RX62x device to the stack.

**Prototype**

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

**Parameters**

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

**Return value**

Reference to the added host controller (0-based index).

### 15.3.5.6 USBH_RX63_Add()

#### Description

Adds a Renesas USB controller of a RX63x device to the stack.

#### Prototype

```
U32 USBH_RX63_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

## 15.3.5.7 USBH_RX64_Add()

### Description

Adds a Renesas USB controller of a RX64x device to the stack.

### Prototype

```
U32 USBH_RX64_Add(void * pBase);
```

### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

### Return value

Reference to the added host controller (0-based index).

## 15.3.5.8 USBH_RX65_Add()

### Description

Adds a Renesas USB controller of a RX65x device to the stack.

### Prototype

```
U32 USBH_RX65_Add(void * pBase);
```

### Parameters

| Parameter | Description |
| --- | --- |
| pBase | Pointer to the base of the controllers register set. |

### Return value

Reference to the added host controller (0-based index).

### 15.3.5.9 USBH_RX71_FS_Add()

#### Description

Adds a Renesas FS USB controller of a RX71x device to the stack.

#### Prototype

```
U32 USBH_RX71_FS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

### 15.3.5.10 USBH_RX71_HS_Add()

#### Description

Adds a Renesas HS USB controller of a RX71x device to the stack.

#### Prototype

```
U32 USBH_RX71_HS_Add(void * pBase);
```

#### Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

# 15.3.5.11 USBH_RZA1_Add()

## Description

Adds a Renesas USB controller of a RZA1 device to the stack.

## Prototype

```
U32 USBH_RZA1_Add(void * pBase);
```

## Parameters

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

## Return value

Reference to the added host controller (0-based index).

### 15.3.5.12 USBH_Synergy_FS_Add()

#### Description

Adds a Renesas FS USB controller of a Synergy device to the stack.

#### Prototype

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

#### Parameters

| Parameter | Description |
| --- | --- |
| pBase | Pointer to the base of the controllers register set. |

#### Return value

Reference to the added host controller (0-based index).

#### 15.3.5.13 USBH_Synergy_HS_Add()

**Description**

Adds a Renesas HS USB controller of a Synergy device to the stack.

**Prototype**

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

**Parameters**

| Parameter | Description |
|---|---|
| pBase | Pointer to the base of the controllers register set. |

**Return value**

Reference to the added host controller (0-based index).

## 15.3.6 ATSAMx7 driver

Using this driver there is no need to provide a separate memory area with `USBH_AssignTransferMemory()`. A single memory heap is sufficient for the USB stack, see `USBH_AssignMemory()`.

### 15.3.6.1 Restrictions

#### HUB support

Although the USB controller of the ATSAMx7 MCUs support external HUBs, the application is very limited. Because USB pipes can not be dynamically allocated to devices, connecting and disconnecting devices arbitrarily to and from the HUB will result in blocked pipes very soon, and new connected devices will not be recognized any more.

Also split transactions are not supported, so low speed and full speed devices can not be used via a high speed HUB.

### 15.3.6.2 ATSAMx7 driver specific configuration functions

| Function | Description |
|---|---|
| `USBH_SAMx7_Add()` | Adds a HS capable ATSAM USBHS controller to the stack. |

### 15.3.6.3 USBH_SAMx7_Add()

#### Description

Adds a HS capable ATSAM USBHS controller to the stack.

#### Prototype

```
U32 USBH_SAMx7_Add(PTR_ADDR BaseAddr,
                   PTR_ADDR USBRAMAddr);
```

#### Parameters

| Parameter | Description |
|---|---|
| BaseAddr | Base address of the USBHS controllers register set. |
| USBRAMAddr | Base address of the USBHS FIFO RAM. |

#### Return value

Reference to the added host controller (0-based index).

## 15.3.7 LPC54xxx High Speed driver

Using this driver there is no need to provide a separate memory area with USBH_AssignTransferMemory(). A single memory heap is sufficient for the USB stack, see USBH_AssignMemory(). The driver uses the dedicated USB1 RAM of the LPC54xxx MCU for endpoint transfer buffers.

### 15.3.7.1 LPC54xxx driver specific configuration functions

| Function | Description |
|---|---|
| USBH_LPC54xxx_Add() | Adds a LPC54xxx high speed controller to the stack. |

### 15.3.7.2 USBH_LPC54xxx_Add()

#### Description

Adds a LPC54xxx high speed controller to the stack.

#### Prototype

```
U32 USBH_LPC54xxx_Add(PTR_ADDR BaseAddr,
                      PTR_ADDR USBRAMAddr,
                      unsigned USBRAMSize);
```

#### Parameters

| Parameter | Description |
|---|---|
| BaseAddr | Base address of the USB controllers register set. |
| USBRAMAddr | Base address of the dedicated USB RAM. |
| USBRAMSize | Size of the USB RAM in bytes. |

#### Return value

Reference to the added host controller (0-based index).

# Chapter 16

# Support

# 16.1 Contacting support

Before contacting support please make sure that you are using the latest version of the emUSB-Host package. Also please check the chapter *Configuring debugging output* on page 29 and run your application with enabled debug support.

If you are a registered emUSB-Host user and you need to contact the emUSB-Host support please send the following information via email to support_emusb@segger.com:

- Your emUSB-Host registration number.
- emUSB-Host version.
- A detailed description of the problem
- The configuration files `USBH_Config*.*`
- Any error messages.

Please also take a few moments to help us to improve our service by providing a short feedback when your support case has been solved.

# Chapter 17

# 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.

## 17.1 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.

> **Note**
>
> It is not advised to enable all log messages as the large amount of output may affect timing.

## 17.2 API functions

| Function | Description |
| --- | --- |
| General functions | |
| USBH_SetLogFilter() | Sets a mask that defines which logging message should be logged. |
| USBH_SetWarnFilter() | Adds an additional filter condition to the mask which specifies the warning messages that should be displayed. |
| USBH_Log() | This function is called by the stack in debug builds with log output. |
| USBH_Warn() | This function is called by the stack in debug builds with log output. |
| USBH_Panic() | Is called if the stack encounters a critical situation. |
| USBH_Logf_Application() | Displays application log information. |
| USBH_Warnf_Application() | Displays application warning information. |
| USBH_sprintf_Application() | A simple sprintf replacement. |

## 17.2.0.1 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);
```

### Parameters

| Parameter | Description |
| --- | --- |
| FilterMask | Specifies which logging messages should be displayed. |

### 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 chapter Message types.

Please note that enabling all log messages (`0xffffffff`) is not necessary, nor is it advised as it will influence the timing greatly.

## 17.2.0.2 USBH_SetWarnFilter()

### Description

Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.

### Prototype

```
void USBH_SetWarnFilter(U32 FilterMask);
```

### Parameters

| Parameter | Description |
| --- | --- |
| FilterMask | Specifies which warning messages should be added to the filter mask. |

### 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 chapter Message types.

### 17.2.0.3 USBH_Log()

#### Description

This function is called by the stack in debug builds with log output. In a release build, this function is not be linked in.

#### Prototype

```
void USBH_Log(const char * s);
```

#### Parameters

| Parameter | Description |
|---|---|
| s | Pointer to a string holding the log message. |

## 17.2.0.4 USBH_Warn()

### Description

This function is called by the stack in debug builds with log output. In a release build, this function is not be linked in.

### Prototype

```
void USBH_Warn(const char * s);
```

### Parameters

| Parameter | Description |
|---|---|
| s | Pointer to a string holding the warning message. |

## 17.2.0.5 USBH_Panic()

### Description

Is called if the stack encounters a critical situation. In a release build, this function is not be linked in.

### Prototype

```
void USBH_Panic(const char * s);
```

### Parameters

| Parameter | Description |
|---|---|
| s | Pointer to a string holding the error message. |

### 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 the error string 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.

## 17.2.0.6 USBH_Logf_Application()

### Description

Displays application log information.

### Prototype

```
void USBH_Logf_Application(const char * sFormat,
                                          ...);
```

### Parameters

| Parameter | Description |
|---|---|
| sFormat | Message string with optional format specifiers. |
| | ... : Optional argument when format specifiers are specified. |

### 17.2.0.7 USBH_Warnf_Application()

#### Description

Displays application warning information.

#### Prototype

```
void USBH_Warnf_Application(const char * sFormat,
                                              ...);
```

#### Parameters

| Parameter | Description |
|---|---|
| sFormat | Message string with optional format specifiers. |
| | ... : Optional argument when format specifiers are specified. |

## 17.2.0.8 USBH_sprintf_Application()

### Description

A simple sprintf replacement.

### Prototype

```
void USBH_sprintf_Application(      char      * pBuffer,
                                    unsigned    BufferSize,
                              const char      * sFormat,
                                                ...);
```

### Parameters

| Parameter | Description |
|---|---|
| pBuffer | Pointer to a user provided buffer. |
| BufferSize | Size of the buffer in bytes. |
| sFormat | Message string with optional format specifiers. |
| | ... : Optional argument when format specifiers are specified. |

# Chapter 18

# 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. This chapter provides descriptions of the functions required to fully support emUSB-Host in multitasking environments.

# 18.1 General information

emUSB-Host includes an OS abstraction layer which should make it possible to use an arbitrary operating system together with emUSB-Host. To adapt emUSB-Host to a new OS one only has to map the functions listed below in section OS layer API functions to the native OS functions. SEGGER took great care when designing this abstraction layer, to make it easy to under- stand and to adapt to different operating systems.

## 18.1.1 Operating system support supplied with this release

In the current version, abstraction layer for embOS is available. Abstraction layers for other operating systems are available upon request.

## 18.2 OS layer API functions

| Function | Description |
|---|---|
| General functions | |
| USBH_OS_Delay() | Blocks the calling task for a given time. |
| USBH_OS_DisableInterrupt() | Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts. |
| USBH_OS_EnableInterrupt() | Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0. |
| USBH_OS_GetTime32() | Return the current system time in ms. |
| USBH_OS_Init() | Initialize (create) all objects required for task synchronization. |
| USBH_OS_DeInit() | Deletes all objects required for task synchronization. |
| USBH_OS_Lock() | This function locks a mutex object, guarding sections of the stack code against other threads. |
| USBH_OS_Unlock() | Unlocks the mutex used by a previous call to USBH_OS_Lock(). |
| USBH_Task synchronization | |
| USBH_OS_SignalNetEvent() | Wakes the USBH_MainTask() if it is waiting for a event or timeout in the function USBH_OS_WaitNetEvent(). |
| USBH_OS_WaitNetEvent() | Blocks until the timeout expires or a USBH-event occurs. |
| USBH_ISRTask synchronization | |
| USBH_OS_SignalISREx() | Wakes the USBH_ISRTask(). |
| USBH_OS_WaitISR() | Blocks until USBH_OS_SignalISR() is called (from ISR). |
| Application task synchronization | |
| USBH_OS_AllocEvent() | Allocates and returns an event object. |
| USBH_OS_FreeEvent() | Releases an object event. |
| USBH_OS_SetEvent() | Sets the state of the specified event object to signalled. |
| USBH_OS_ResetEvent() | Sets the state of the specified event object to non-signalled. |
| USBH_OS_WaitEvent() | Wait for the specific event. |
| USBH_OS_WaitEventTimed() | Wait for the specific event within a given timeout. |

## 18.2.0.1 USBH_OS_Delay()

### Description

Blocks the calling task for a given time.

### Prototype

```
void USBH_OS_Delay(unsigned ms);
```

### Parameters

| Parameter | Description |
| --- | --- |
| ms | Delay in milliseconds. |

### 18.2.0.2 USBH_OS_DisableInterrupt()

#### Description

Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts.

#### Prototype

```
void USBH_OS_DisableInterrupt(void);
```

#### Additional information

The USB stack will perform nested calls to `USBH_OS_DisableInterrupt()` and `USBH_OS_EnableInterrupt()`.

# 18.2.0.3 USBH_OS_EnableInterrupt()

## Description

Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0.

## Prototype

```
void USBH_OS_EnableInterrupt(void);
```

## Additional information

The USB stack will perform nested calls to `USBH_OS_DisableInterrupt()` and `USBH_OS_EnableInterrupt()`.

# 18.2.0.4 USBH_OS_GetTime32()

### Description

Return the current system time in ms. The value will wrap around after app. 49.7 days. This is taken into account by the stack.

### Prototype

```
U32 USBH_OS_GetTime32(void);
```

### Return value

Current system time.

## 18.2.0.5 USBH_OS_Init()

### Description

Initialize (create) all objects required for task synchronization.

### Prototype

```
void USBH_OS_Init(void);
```

### 18.2.0.6 USBH_OS_DeInit()

#### Description

Deletes all objects required for task synchronization.

#### Prototype

```
void USBH_OS_DeInit(void);
```

## 18.2.0.7 USBH_OS_Lock()

### Description

This function locks a mutex object, guarding sections of the stack code against other threads. Mutexes are recursive.

### Prototype

```
void USBH_OS_Lock(unsigned Idx);
```

### Parameters

| Parameter | Description |
|---|---|
| Idx | Index of the mutex to be locked (0 .. USBH_MUTEX_COUNT-1). |

## 18.2.0.8 USBH_OS_Unlock()

### Description

Unlocks the mutex used by a previous call to USBH_OS_Lock(). Mutexes are recursive.

### Prototype

```
void USBH_OS_Unlock(unsigned Idx);
```

### Parameters

| Parameter | Description |
|---|---|
| Idx | Index of the mutex to be released (0 .. USBH_MUTEX_COUNT-1). |

# 18.2.0.9 USBH_OS_SignalNetEvent()

## Description

Wakes the USBH_MainTask() if it is waiting for a event or timeout in the function USBH_OS_WaitNetEvent().

## Prototype

```
void USBH_OS_SignalNetEvent(void);
```

### 18.2.0.10 USBH_OS_WaitNetEvent()

#### Description

Blocks until the timeout expires or a USBH-event occurs. Called from `USBH_MainTask()` only. A USBH-event is signaled with `USBH_OS_SignalNetEvent()` called from an other task or ISR.

#### Prototype

```
void USBH_OS_WaitNetEvent(unsigned ms);
```

#### Parameters

| Parameter | Description |
| --- | --- |
| ms | Timeout in milliseconds. |

## 18.2.0.11 USBH_OS_SignalISREx()

### Description

Wakes the `USBH_ISRTask()`. Called from ISR.

### Prototype

```
void USBH_OS_SignalISREx(U32 DevIndex);
```

### Parameters

| Parameter | Description |
|---|---|
| DevIndex | Zero-based index of the host controller that needs attention. |

### 18.2.0.12 USBH_OS_WaitISR()

#### Description

Blocks until `USBH_OS_SignalISR()` is called (from ISR). Called from `USBH_ISRTask()` only.

#### Prototype

```
U32 USBH_OS_WaitISR(void);
```

#### Return value

An ISR mask, where each bit set corresponds to a host controller index.

## 18.2.0.13 USBH_OS_AllocEvent()

### Description

Allocates and returns an event object.

### Prototype

```
USBH_OS_EVENT_OBJ *USBH_OS_AllocEvent(void);
```

### Return value

A pointer to a USBH_OS_EVENT_OBJ object on success or NULL on error.

## 18.2.0.14 USBH_OS_FreeEvent()

### Description

Releases an object event.

### Prototype

```
void USBH_OS_FreeEvent(USBH_OS_EVENT_OBJ * pEvent);
```

### Parameters

| Parameter | Description |
|---|---|
| pEvent | Pointer to an event object that was returned by USBH_OS_AllocEvent(). |

### 18.2.0.15 USBH_OS_SetEvent()

#### Description

Sets the state of the specified event object to signalled.

#### Prototype

```
void USBH_OS_SetEvent(USBH_OS_EVENT_OBJ * pEvent);
```

#### Parameters

| Parameter | Description |
|---|---|
| pEvent | Pointer to an event object that was returned by USBH_OS_AllocEvent(). |

## 18.2.0.16 USBH_OS_ResetEvent()

### Description

Sets the state of the specified event object to non-signalled.

### Prototype

```
void USBH_OS_ResetEvent(USBH_OS_EVENT_OBJ * pEvent);
```

### Parameters

| Parameter | Description |
|---|---|
| pEvent | Pointer to an event object that was returned by USBH_OS_AllocEvent(). |

## 18.2.0.17 USBH_OS_WaitEvent()

### Description

Wait for the specific event.

### Prototype

```
void USBH_OS_WaitEvent(USBH_OS_EVENT_OBJ * pEvent);
```

### Parameters

| Parameter | Description |
|---|---|
| pEvent | Pointer to an event object that was returned by USBH_OS_AllocEvent(). |

## 18.2.0.18 USBH_OS_WaitEventTimed()

### Description

Wait for the specific event within a given timeout.

### Prototype

```
int USBH_OS_WaitEventTimed(USBH_OS_EVENT_OBJ * pEvent,
                           U32                 MilliSeconds);
```

### Parameters

| Parameter | Description |
| --- | --- |
| pEvent | Pointer to an event object that was returned by USBH_OS_AllocEvent(). |
| MilliSeconds | Timeout in milliseconds. |

### Return value

| | |
| --- | --- |
| USBH_OS_EVENT_SIGNALED | Event was signaled. |
| USBH_OS_EVENT_TIMEOUT | Timeout occurred. |

# Chapter 19

# 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.

# 19.1 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. The code size depend on the API functions called by the application. The code was compiled for a Cortex-M4 CPU with size optimization. Note that the values are only valid for an average configuration.

## 19.1.1 ROM

The following table shows the approximate ROM requirement of emUSB-Host (compiled with gcc -Os):

| Component | ROM |
|---|---|
| USB core | 6.7 KBytes |
| HUB Support | 3.2 KBytes |
| CDC | 5.4 KBytes |
| Vendor class | 3.9 KBytes |
| FT232 | 3.9 KBytes |
| HID Generic | 6.1 KBytes |
| HID Mouse Keyboard | 6.8 KBytes |
| MSD | 6.9 KBytes + sizeof(Filesystem) |
| MTP | 13.2 KBytes |
| Printer | 3.1 KBytes |
| LAN using ASIX | 7.2 KBytes + sizeof(embOS/IP) |
| LAN using CDC-ECM | 7.8 KBytes + sizeof(embOS/IP) |
| LAN using RNDIS | 8.1 KBytes + sizeof(embOS/IP) |
| Driver EHCI | 5.0 KBytes |
| Driver OHCI | 7.6 KBytes |
| Driver STM32F4 FS | 4.3 KBytes |
| Driver STM32F4 HS | 4.7 KBytes |
| Driver STM32F7 HS | 4.8 KBytes |
| Driver Kinetis FS | 3.1 KBytes |
| Driver Renesas RX64 | 4.7 KBytes |

## 19.1.2 RAM

The following table shows the average RAM requirement of emUSB-Host. The actual RAM usage may vary depending on the USB host controller used, the memory architecture of the target, the USB devices connected to emUSB-Host and the type of operations performed with that devices.

| Component | RAM |
|---|---|
| emUSB-Host core incl. one driver | 3.8 KByte |
| For each connected generic HID device | 3.0 KByte |
| For each connected CDC ACM device | 4.0 KByte |
| For each connected Vendor (BULK) device | 3.5 KByte |
| For each connected MSD device | 2.2 KByte |
| For each connected Mouse | 4.6 KByte |
| For each connected external HUB | 2.0 KByte |
| For each connected LAN (ASIX) device | 11.1 KByte |

| Component | RAM |
| --- | --- |
| For each connected LAN (CDC-ECM) device | 18.1 KByte |
| For each connected LAN (RNDIS) device | 13.5 KByte |

## 19.2 Performance

The following values have been tested using the CDC protocol. An emPower evaluation board running emUSB-Device CDC class was connected to the host.

System with Synopsys (USB High-Speed) controller:

| Description | Speed |
|---|---|
| Send speed | 34.9 MByte/sec |
| Receive speed | 39.0 MByte/sec |

System with EHCI (USB High-Speed) controller:

| Description | Speed |
|---|---|
| Send speed | 30.9 MByte/sec |
| Receive speed | 36.0 MByte/sec |

System with OHCI (USB Full-Speed) controller:

| Description | Speed |
|---|---|
| Send speed | 800 KByte/sec |
| Receive speed | 800 KByte/sec |

# Chapter 20

# Glossary

| Term | Definition |
| --- | --- |
| BSP | Board support package. |
| CPU | Central Processing Unit. The “brain” of a microcontroller; the part of a processor that carries out instructions. |
| DMA | Direct Memory Access. |
| 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). |
| MCU | Microcontroller unit. |
| MMU | Memory managing unit |
| NULL packet | See ZLP. |
| PLL | Phase-locked loop, used for clock generation inside a microcontroller. |
| RTOS | Real-time Operating System. |
| RTT | Real-Time Transfer. Method to output information from the target microcontroller as well as sending input to the application at a very high speed without affecting the target’s real time behavior. |
| 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. |
| ZLP | Zero-Length-Packet |

# Chapter 21

# FAQ

This chapter answers some frequently asked questions.

Q: **Which CPUs can I use emUSB-Host 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: Not generally. Different devices or endpoints can be handled in different tasks without restrictions. For example a task may read from one endpoint or device while another task is writing to another endpoint or device. If accessing the same resource, the user is responsible for locking 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 write 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. Also consider restrictions listed in *Host controller specifics* on page 375.

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.