

# ***emUSB-Host***

CPU independent  
USB Host stack for  
embedded applications

## User Guide

Software version 1.00

Document: UM10001

Revision: 0

Date: June 19, 2009



A product of SEGGER Microcontroller GmbH & Co. KG

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

## **Disclaimer**

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

## **Copyright notice**

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of the manufacturer. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2009 SEGGER Microcontroller GmbH & Co. KG, Hilden / Germany

## **Trademarks**

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

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

## **Contact address**

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11  
D-40721 Hilden

Germany

Tel. +49 2103-2878-0

Fax. +49 2103-2878-28

Email: [support@segger.com](mailto:support@segger.com)

Internet: <http://www.segger.com>

## Manual versions

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

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

SW version / Manual revision	Date	By	Explanation
1.00/00	090609	AS	Initial version.

## Software versions

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



# About this document

---

## Assumptions

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

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

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

## How to use this manual

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

## Typographic conventions for syntax

This manual uses the following typographic conventions:

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

**Table 1.1: Typographic conventions**



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

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

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

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

**Corporate Office:**

<http://www.segger.com>

**United States Office:**

<http://www.segger-us.com>

## EMBEDDED SOFTWARE (Middleware)



**emWin**

**Graphics software and GUI**

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.



**embOS**

**Real Time Operating System**

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.



**emFile**

**File system**

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.



**USB-Stack**

**USB device stack**

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

## SEGGER TOOLS

**Flasher**

**Flash programmer**

Flash Programming tool primarily for microcontrollers.

**J-Link**

**JTAG emulator for ARM cores**

USB driven JTAG interface for ARM cores.

**J-Trace**

**JTAG emulator with trace**

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

**J-Link / J-Trace Related Software**

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.



# Table of Contents

1	Introduction to emUSB-Host .....	9
1.1	What is emUSB-Host.....	10
1.2	Features.....	10
1.3	Basic concepts .....	11
1.4	Further reading .....	12
1.5	Development environment (compiler).....	13
2	Running emUSB-Host on target hardware .....	15
2.1	Step 1: Open an emUSB-Host start project .....	17
2.2	Step 2: Adding emUSB-Host to the start project .....	18
2.3	Step 3: Build the project and test it .....	19
3	Example applications .....	21
3.1	Overview .....	22
4	USB Host Core Functions.....	23
4.1	Management Functions.....	24
4.2	API Functions .....	28
4.3	Data Structures .....	52
4.4	Function Types.....	75
4.5	Use of undocumented functions .....	79
5	Mass Storage Device.....	81
5.1	Introduction.....	82
5.1.1	Overview .....	82
5.1.2	Features.....	82
5.1.3	Restrictions.....	82
5.1.4	Requirements.....	82
5.1.5	Example code .....	82
5.1.6	Supported Protocols.....	82
5.2	API Functions .....	84
5.2.1	USBH_MSD_Init().....	85
5.3	Data Structures .....	91
5.4	Function Types.....	93
5.5	Error Codes .....	95
5.5.1	USBH_MSD_STATUS_ERROR.....	96
5.5.2	USBH_MSD_STATUS_PARAMETER.....	96
5.5.3	USBH_MSD_STATUS_LENGTH .....	96
5.5.4	USBH_MSD_STATUS_TIMEOUT .....	96
5.5.5	USBH_MSD_STATUS_COMMAND_FAILED.....	96
5.5.6	USBH_MSD_STATUS_INTERFACE_PROTOCOL.....	96
5.5.7	USBH_MSD_STATUS_INTERFACE_SUB_CLASS .....	96
5.5.8	USBH_MSD_STATUS_PIPE_STALLED .....	96
5.5.9	USBH_MSD_STATUS_TRANSMISSION .....	96
5.5.10	USBH_MSD_STATUS_SENSE_STOP.....	97
5.5.11	USBH_MSD_STATUS_SENSE_REPEAT.....	97
5.5.12	USBH_MSD_STATUS_WRITE_PROTECT .....	97
6	Human Interface Device .....	99

6.1	Introduction .....	100
6.1.1	Overview .....	100
6.1.2	Restrictions .....	100
6.1.3	Example code .....	100
6.2	API Functions .....	101
6.2.1	USBH_HID_Init() .....	102
6.2.2	USBH_HID_SetOnMouseStateChange() .....	104
6.2.3	USBH_HID_SetOnKeyboardStateChange() .....	105
6.3	Data Structures .....	106
6.4	Function Types .....	109
7	Configuring emUSB-Host.....	113
7.1	Runtime configuration.....	114
7.1.1	Driver handling .....	114
7.2	Compile-time configuration.....	115
7.2.1	Compile-time configuration switches .....	115
7.2.2	Debug level .....	115
8	Debugging.....	117
8.1	Message output .....	118
8.2	API functions .....	119
8.3	Message types .....	127
9	OS integration .....	129
9.1	General information.....	130
9.2	OS layer API functions .....	131
9.2.1	Examples .....	131
10	Performance & resource usage .....	133
10.1	Memory footprint .....	134
10.1.1	ROM .....	134
10.1.2	RAM .....	134
10.2	Performance.....	135
11	Related Documents .....	137
12	Glossary .....	139

# Chapter 1

## Introduction to emUSB-Host

---

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

## 1.1 What is emUSB-Host

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

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

## 1.2 Features

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

- 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

## 1.4 Further reading

This guide explains the usage of the emUSB-Host protocol stack. It describes all functions which are required to build a network application. For a deeper understanding about how the USB protocols works use the following references.

## 1.5 Development environment (compiler)

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

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

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

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



# Chapter 2

## Running emUSB-Host on target hardware

---

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

## Integrating emUSB-Host

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

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

### Procedure to follow

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

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

## 2.1 Step 1: Open an emUSB-Host start project

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

## 2.2 Step 2: Adding emUSB-Host to the start project

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

- Config
- USBH

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

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

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

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

Replace the `BSP.c` source file and the `BSP.h` header file used in your emUSB-Host start project with the one which is supplied with the emUSB-Host shipment. Some drivers require a special functions which initializes the USB Host interface. This function is called `BSP_USBH_Init()`. It is used to enable the ports which are connected to the hardware. All interface driver packages include the `BSP.c` and `BSP.h` files irrespective if the `BSP_USBH_Init()` function is implemented.

### 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 directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- Config
- Inc
- USBH

### Select the start application

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

## 2.3 Step 3: Build the project and test it

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



# Chapter 3

## Example applications

---

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

## 3.1 Overview

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

The following start application files are provided:

File	Description
------	-------------

**Table 3.1: emUSB-Host example applications**

The example applications for the target-side are supplied in source code in the Application directory.

# Chapter 4

## USB Host Core Functions

---

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

## 4.1 Management Functions

The table below lists the available management functions.

Function	Description
<code>USBH_Init()</code>	Basically initializes the USB Host stack.
<code>USBH_Exit()</code>	Is called on exit of the library.
<code>USBH_EnumerateDevices()</code>	Adds default endpoints for enumeration, sets the host controller into running state and starts the enumeration of the complete bus.

**Table 4.1: emUSB-Host management function overview**

## 4.1.1 USBH\_Init()

### Description

Basically initializes the USB Host stack.

### Prototype

```
USBH_STATUS USBH_Init();
```

### Additional information

Has to be called one time during startup before any other function. The library initializes or allocates global resources within this function. The host controller must be created and added to the bus driver at a later time.

## 4.1.2 USBH\_Exit()

### Description

Is called on exit of the library

### Prototype

```
void USBH_Exit();
```

### Additional information

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

### 4.1.3 USBH\_EnumerateDevices()

#### Description

Adds default endpoints for enumeration, sets the host controller into running state and starts the enumeration of the complete bus.

#### Prototype

```
void USBH_EnumerateDevices( USBH_HC_BD_HANDLE * HcBdHandle );
```

#### Additional information

If this function returns the host controller runs and can detect USB devices.

## 4.2 API Functions

The table below lists the available API functions.

Function	Description
<code>USBH_CreateInterfaceList()</code>	Generates a list of available interfaces.
<code>USBH_DestroyInterfaceList()</code>	Deletes a previously generated interface list.
<code>USBH_GetInterfaceID()</code>	Returns the interface ID for a specified interface.
<code>USBH_GetInterfaceInfo()</code>	Obtains information about a specified interface.
<code>USBH_RegisterPnPNotification()</code>	Registers a notification function for PnP events.
<code>USBH_UnregisterPnPNotification()</code>	Unregisters a previously registered notification for PnP events.
<code>USBH_RegisterEnumErrorNotification()</code>	Registers a port error enumeration notification.
<code>USBH_UnregisterEnumErrorNotification()</code>	Unregisters an registered port error enumeration notification.
<code>USBH_RestartEnumError()</code>	The enumeration for all devices that have failed the enumeration is restarted.
<code>USBH_OpenInterface()</code>	Opens the specified interface.
<code>USBH_CloseInterface()</code>	Closes a previously opened interface.
<code>USBH_GetDeviceDescriptor()</code>	Retrieves the device descriptor.
<code>USBH_GetCurrentConfigurationDescriptor()</code>	Retrieves the current configuration descriptor.
<code>USBH_GetInterfaceDescriptor()</code>	Retrieves the interface descriptor.
<code>USBH_GetEndpointDescriptor()</code>	Retrieves a endpoint descriptor.
<code>USBH_GetSerialNumber()</code>	Retrieves the serial number.
<code>USBH_GetSpeed()</code>	Retrieves the operation speed of the device.
<code>USBH_GetFrameNumber()</code>	Retrieves the current frame number.
<code>USBH_GetInterfaceIDByHandle()</code>	Retrieves the current frame number.
<code>USBH_SubmitUrb()</code>	Is used to submit an URB.
<code>USBH_SetTraceMask()</code>	Sets an internal trace mask which filters trace messages produced by the USB bus driver.
<code>USBH_GetStatusStr()</code>	Return the status as an string constants.

**Table 4.2: emUSB-Host API function overview**

## 4.2.1 USBH\_CreateInterfaceList()

### Description

Generates a list of available interfaces.

### Prototype

```
USBH_INTERFACE_LIST_HANDLE USBH_CreateInterfaceList (
    USBH_INTERFACE_MASK * InterfaceMask,
    unsigned int         * InterfaceCount );
```

### Parameters

Parameter	Description
<a href="#">InterfaceMask</a>	Input parameter of type USBH_INTERFACE_MASK which specifies a mask for the interfaces which should be listed.
<a href="#">InterfaceCount</a>	Returns the number of available interfaces.

**Table 4.3: USBH\_CreateInterfaceList() parameter list**

### Additional information

The generated interface list is stored in the bus driver and must be deleted by a call to `USBH_DestroyInterfaceList`. The list contains a snap shoot 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.

## 4.2.2 USBH\_DestroyInterfaceList()

### Description

Deletes a previously generated interface list.

### Prototype

```
void USBH_DestroyInterfaceList(  
    USBH_INTERFACE_LIST_HANDLE InterfaceListHandle );
```

### Parameter

Parameter	Description
<a href="#">InterfaceListHandle</a>	Contains the handle for the interface list. It must not be NULL.

**USBH\_DestroyInterfaceList() parameter list**

### Additional information

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

## 4.2.3 USBH\_GetInterfaceID()

### Description

Returns the interface ID for a specified interface.

### Prototype

```
USBH_INTERFACE_ID USBH_GetInterfaceID(
    USBH_INTERFACE_LIST_HANDLE InterfaceListHandle,
    unsigned int Index );
```

### Parameters

Parameter	Description
<a href="#">InterfaceListHandle</a>	Contains the handle for the interface list generated by a call to <code>USBH_CreateInterfaceList</code> .
<a href="#">Index</a>	Specifies the zero based index for an interface in the list.

#### USBH\_GetInterfaceID() parameter list

### Return value

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

### Additional information

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

## 4.2.4 USBH\_GetInterfaceInfo()

### Description

Obtains information about a specified interface.

### Prototype

```
USBH_STATUS USBH_GetInterfaceInfo(  
    USBH_INTERFACE_ID    InterfaceID,  
    USBH_INTERFACE_INFO * InterfaceInfo );
```

### Return value

Returns USBH\_STATUS\_SUCCESS or USBH\_STATUS\_DEVICE\_REMOVED.

### Additional information

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

## 4.2.5 USBH\_RegisterPnPNotification()

### Description

Registers a notification function for PnP events.

### Prototype

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

### Parameter

Parameter	Description
<code>PnPNotification</code>	Contains a pointer to a caller provided structure <code>USBH_PNP_NOTIFICATION</code> . This structure must be filled in by the caller.

**Table 4.4: USBH\_RegisterPnPNotification() parameter list**

### Return value

On success a valid handle is returned, or NULL on error.

### Additional information

If a valid handle is returned, the function `USBH_UnregisterPnPNotification` must be called to release the notification. An application can register any number of notifications. The user notification routine is called in the context of an 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 function `USBH_PnpNotification` is called for each matching interface.

## 4.2.6 USBH\_UnregisterPnPNotification()

### Description

Unregisters a previously registered notification for PnP events.

### Prototype

```
void USBH_UnregisterPnPNotification( USBH_NOTIFICATION_HANDLE Handle );
```

### Parameter

Parameter	Description
<a href="#">Handle</a>	Contains the valid handle for a PnP notification previously registered by a call to <code>USBH_RegisterPnPNotification</code> .

**Table 4.5: USBH\_UnregisterPnPNotification() parameter list**

### Additional information

Has to be called for a PnP notification that was successfully registered by a call to `USBH_RegisterPnPNotification`.

## 4.2.7 USBH\_RegisterEnumErrorNotification()

### Description

Registers a port error enumeration notification.

### Prototype

```
USBH_ENUM_ERROR_HANDLE USBH_RegisterEnumErrorNotification(
    void * Context,
    USBH_EnumErrorNotification * EnumErrorCallback );
```

### Parameters

Parameter	Description
<code>Context</code>	Is a user defined pointer that is passed unchanged to the notification callback function <code>USBH_EnumErrorNotification</code> .
<code>EnumErrorCallback</code>	Contains the notification function that is called from the library if a port enumeration error occurs.

**Table 4.6: USBH\_RegisterEnumErrorNotification() parameter list**

### Return value

On success a valid handle is returned, or NULL on error.

### Additional information

If a valid handle is returned, the function `USBH_RegisterEnumErrorNotification` must be called to release the notification. The `EnumErrorCallback` callback routine is called in the context of the process where the interrupt status of a host controller is processed. It is forbidden to wait in that context.

## 4.2.8 USBH\_UnregisterEnumErrorNotification()

### Description

Unregisters an registered port error enumeration notification.

### Prototype

```
void USBH_UnregisterEnumErrorNotification( USBH_ENUM_ERROR_HANDLE Handle );
```

### Parameter

Parameter	Description
Handle	Contains the valid handle for the notification previously returned from <code>USBH_RegisterEnumErrorNotification</code> .

**Table 4.7: USBH\_UnregisterEnumErrorNotification() parameter list**

### Additional information

Has to be called for a port enumeration error notification that was successfully registered by a call to `USBH_RegisterEnumErrorNotification`.

## 4.2.9 USBH\_RestartEnumError()

### Description

The enumeration for all devices that have failed the enumeration is restarted.

### Prototype

```
void USBH_RestartEnumError();
```

### Additional information

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

## 4.2.10 USBH\_OpenInterface()

### Description

Opens the specified interface.

### Prototype

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

### Parameters

Parameter	Description
<code>InterfaceID</code>	Specifies the interface to open by its interface ID. The interface ID can be obtained by <code>USBH_PnpNotification</code> or <code>USBH_GetInterfaceID</code> .
<code>Exclusive</code>	Specifies if the interface should be opened exclusive or not. If the value is unequal of zero the interface is opened exclusive.
<code>InterfaceHandle</code>	Returns the handle for the opened interface on success.

**Table 4.8: USBH\_OpenInterface() parameter list**

### Return value

Returns success or an error. The function can fail if the device is or was removed or the device is opened exclusive by a different application. The function returns with error if the exclusive flag is true and a different application has an open handle to the function.

### Additional information

The handle returned by this function is used by all other function that perform a data transfer. The returned handle must be closed with `USBH_CloseInterface` if it is no longer required

## 4.2.11 USBH\_CloseInterface()

### Description

Closes the specified interface.

### Prototype

```
void USBH_CloseInterface( USBH_INTERFACE_HANDLE Handle );
```

### Parameter

Parameter	Description
Handle	Contains the handle for an interface opened by a call to <code>USBH_OpenInterface</code> . It must not be NULL.

**Table 4.9: USBH\_CloseInterface() parameter list**

### Additional information

Each handle must be closed one time. The library access invalid memory if this function is called with an invalid handle.

## 4.2.12 USBH\_GetDeviceDescriptor()

### Description

Retrieves the device descriptor.

### Prototype

```
USBH_STATUS USBH_GetDeviceDescriptor(
    USBH_INTERFACE_HANDLE Handle,
    U8 * Descriptor,
    unsigned int Size,
    unsigned int * Count );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
Descriptor	Points to a caller provided buffer that retrieves the device descriptor on success.
Size	Specifies the size of the caller provided buffer.
Count	Returns the length of the returned descriptor.

**Table 4.10: USBH\_GetDeviceDescriptor() parameter list**

### Return value

Success or device removed.

### Additional information

Returns a copy of the device descriptor and does not access the device. If the buffer is smaller than the device descriptor the function returns the first part of it.

## 4.2.13 USBH\_GetCurrentConfigurationDescriptor()

### Description

Retrieves the current configuration descriptor.

### Prototype

```
USBH_STATUS USBH_GetCurrentConfigurationDescriptor (
    USBH_INTERFACE_HANDLE  Handle,
    U8                     * Descriptor,
    unsigned int           Size,
    unsigned int           * Count );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
Descriptor	Points to a caller provided buffer that retrieves the current configuration descriptor on success.
Size	Specifies the size of the caller provided buffer.
Count	Returns the number of valid bytes.

**Table 4.11: USBH\_GetCurrentConfigurationDescriptor() parameter list**

### Return value

Success or device removed.

### Additional information

Returns a copy of the current configuration descriptor. The descriptor is a copy that was stored during the device enumeration. The function returns the first part of the descriptor if the buffer is smaller than the descriptor. This descriptor contains all interface, endpoint, and possible class descriptors. The size is variable. The current configuration descriptor is the descriptor return to the request with the index 0 if the device was enumerated by the device the first time. It changes if the configuration is switch with `USBH_SET_CONFIGURATION`. Other configuration descriptors of a multi-configuration device can be requested with `USBH_FUNCTION_CONTROL_REQUEST`.

## 4.2.14 USBH\_GetInterfaceDescriptor()

### Description

Retrieves the interface descriptor.

### Prototype

```
USBH_STATUS USBH_GetInterfaceDescriptor(
    USBH_INTERFACE_HANDLE Handle,
    U8 AlternateSetting,
    U8 * Descriptor,
    unsigned int Size,
    unsigned int * Count );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
AlternateSetting	Specifies the alternate setting for this interface.
Descriptor	Points to a caller provided buffer that retrieves the interface descriptor on success.
Size	Specifies the size of the caller provided buffer.
Count	Returns the number of valid bytes in the descriptor.

**Table 4.12: USBH\_GetInterfaceDescriptor() parameter list**

### Return value

Success, device removed, or invalid parameter.

### Additional information

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

## 4.2.15 USBH\_GetEndpointDescriptor()

### Description

Retrieves a endpoint descriptor.

### Prototype

```

USBH_STATUS USBH_GetEndpointDescriptor(
    USBH_INTERFACE_HANDLE  Handle,
    U8                     AlternateSetting,
    USBH_EP_MASK           * Mask,
    U8                     * Descriptor,
    unsigned int           Size,
    unsigned int           * Count );

```

### Parameters

Parameter	Description
<code>Handle</code>	Specifies the interface by its interface handle.
<code>AlternateSetting</code>	Specifies the alternate setting for the interface. The function returns endpoint descriptors that are inside the specified alternate setting.
<code>Mask</code>	Is of type <code>USBH_EP_MASK</code> and specifies a mask to select the endpoint.
<code>Descriptor</code>	Returns a pointer to a caller provided buffer that contains the endpoint descriptor on success.
<code>Size</code>	Specifies the size of the caller provided buffer.
<code>Count</code>	Returns the valid number of bytes written to the buffer.

**Table 4.13: USBH\_GetEndpointDescriptor() parameter list**

### Return value

Fails if the endpoint cannot be found or if the device is removed.

### Additional information

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

## 4.2.16 USBH\_GetSerialNumber()

### Description

Retrieves the serial number.

### Prototype

```
USBH_STATUS USBH_GetSerialNumber(
    USBH_INTERFACE_HANDLE Handle,
    U8 * Descriptor,
    unsigned int Size,
    unsigned int * Count );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
Descriptor	Is a pointer to a caller provided buffer. It returns the serial number on success.
Size	Specifies the size of the caller provided buffer in bytes.
Count	Returns the number of bytes written to the buffer.

**Table 4.14: USBH\_GetSerialNumber() parameter list**

### Return value

Returns an error if the device is removed.

### Additional information

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

## 4.2.17 USBH\_GetSpeed()

### Description

Retrieves the operation speed of the device.

### Prototype

```
USBH_STATUS USBH_GetSpeed(
    USBH_INTERFACE_HANDLE Handle,
    USBH_SPEED * Speed );
```

### Parameters

Parameter	Description
<a href="#">Handle</a>	Specifies the interface by its interface handle.
<a href="#">Speed</a>	Returns the operating speed of the device. It is of type USBH_SPEED.

**Table 4.15: USBH\_GetSpeed() parameter list**

### Return value

Returns an error if the device is removed.

### Additional information

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

## 4.2.18 USBH\_GetFrameNumber()

### Description

Retrieves the current frame number.

### Prototype

```
USBH_STATUS USBH_GetFrameNumber (
    USBH_INTERFACE_HANDLE  Handle,
    U32                    * FrameNumber );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
FrameNumber	Returns the current frame number on success.

**Table 4.16: USBH\_GetFrameNumber() parameter list**

### Return value

Returns an error if the device is removed.

### Additional information

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

## 4.2.19 USBH\_GetInterfaceIDByHandle()

### Description

Retrieves the interface ID for a given interface.

### Prototype

```
USBH_STATUS USBH_GetInterfaceIDByHandle(
    USBH_INTERFACE_HANDLE Handle,
    USBH_INTERFACE_ID * InterfaceID );
```

### Parameters

Parameter	Description
<a href="#">Handle</a>	Specifies the interface by its interface handle.
<a href="#">InterfaceID</a>	Returns the interface ID on success.

**Table 4.17: USBH\_GetInterfaceIDByHandle() parameter list**

### Return value

Returns an error if the device is removed.

### Additional information

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

## 4.2.20 USBH\_SubmitUrb()

### Description

Is used to submit an URB.

### Prototype

```
USBH_STATUS USBH_SubmitUrb(
    USBH_INTERFACE_HANDLE Handle,
    URB * Urb );
```

### Parameters

Parameter	Description
Handle	Specifies the interface by its interface handle.
Urb	Input and output parameter. On input it contains the URB which should be submitted. On output it contains the submitted URB with the appropriate status and the received data if any. The storage for the URB must be permanent as long as the request is pending. The host controller can define special alignment requirements for the URB or the data transfer buffer.

**Table 4.18: USBH\_SubmitUrb() parameter list**

### Return value

The request can fail on different reasons. 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 success, the request was processed immediately. On error the request cannot be processed.

### Additional information

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

## 4.2.21 USBH\_SetTraceMask()

### Description

Sets an internal trace mask which filters trace messages produced by the USB bus driver

### Prototype

```
void USBH_SetTraceMask( U32 Mask );
```

### Parameter

Parameter	Description
Mask	Specifies the new trace mask to be set.

**Table 4.19: USBH\_SetTraceMask() parameter list**

### Additional information

The trace mask is an internal global integer variable. A specific bit position within that variable is assigned to every particular trace message built into the USB bus library. The message will be outputted if the corresponding bit is set and will be suppressed if the corresponding bit is cleared. This way, the current value of the trace mask determines the amount of trace messages produced by the USB bus library.

Bit positions of trace mask are assigned as described below:

DBG\_ERR

Fatal errors and ASSERTs. It is recommended to always set this bit.

DBG\_WRN

Non-fatal errors. Warning messages. It is recommended to always set this bit.

DBG\_INFO

Informational messages.

DBG\_FUNC

Function names.

DBG\_UPPER

Print functions names with parameters of the upper interface.

DBG\_EP

Endpoint object and interface traces.

DBG\_EP0

Control endpoint object traces.

DBG\_HOST

Host object traces.

DBG\_RHUB

RootHub object traces.

DBG\_DRV

Driver object traces.

DBG\_PNP

PNP notification traces.

DBG\_DEV

Device object traces.

DBG\_URBCT

Traces an URB counter for testing.

DBG\_HUB

Hub object traces.

DBG\_REFCT

Trace an internal reference counter.

**DBG\_SUBSTATE**

Trace an helper sub state machine.

**DBG\_HUBNOTIFY**

Trace hub device status notifications.

**DBG\_ADDREMOVE**

Display informations about adding and removing of USB devices.

By default, the bits `DBG_ERR` and `DBG_WRN` are set and all other bits are cleared in the trace mask. Note that the `DBG_xxx` constants specify a bit position and not the corresponding mask. Use the `DBG_BIT_MASK` macro to create the corresponding mask for an individual bit position.

**Example:**

```
USBH_SetTraceMask (DBG_BIT_MASK (DBG_ERR) | DBG_BIT_MASK (DBG_WRN) ) ;
```

In the debug version the trace support is enabled. If trace support is disabled then a call to `USBH_SetTraceMask` has no effect.

## 4.2.22 USBH\_GetStatusStr()

### Description

Return the status as an string constants.

### Prototype

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

### Parameter

Parameter	Description
<a href="#">x</a>	Specifies the status.

**Table 4.20: USBH\_GetStatusStr() parameter list**

### Return value

An error string is returned.

### Additional information

Returns only an error string if the debug version of the library is used (DBG=1).

## 4.3 Data Structures

The table below lists the available data structures.

Structure	Description
<code>USBH_INTERFACE_MASK</code>	Input parameter to create an interface list or to register a PnP notification.
<code>USBH_INTERFACE_INFO</code>	Contains information about a USB interface and the related device.
<code>USBH_ENUM_ERROR</code>	Is used as an notification parameter for the <code>USBH_EnumErrorNotification</code> function.
<code>USBH_EP_MASK</code>	Input parameter to get an endpoint descriptor.
<code>USBH_CONTROL_REQUEST</code>	Is used as a union member for the URB data structure.
<code>USBH_BULK_INT_REQUEST</code>	Is used to transfer data from or to a bulk endpoint.
<code>USBH_ISO_FRAME</code>	Is used to define ISO transfer buffers.
<code>USBH_ISO_REQUEST</code>	Is used to transfer data to an ISO endpoint.
<code>USBH_ENDPOINT_REQUEST</code>	Is used as a union member for the URB data structure.
<code>USBH_SET_CONFIGURATION</code>	Is used as a union member for the URB data structure.
<code>USBH_SET_INTERFACE</code>	Is used as a union member for the URB data structure.
<code>USBH_SET_POWER_STATE</code>	Is used to set a power state.
<code>URB</code>	Basic structure for all asynchronous operations on the bus driver.
<code>USBH_PNP_NOTIFICATION</code>	Is used as an input parameter for the <code>USBH_RegisterPnPNotification</code> function.
<code>USBH_HEADER</code>	Defines the header of an URB.
<code>USBH_SPEED</code>	Is used to get the operation speed of a device.
<code>USBH_PNP_EVENT</code>	Is used as a parameter for the PnP notification.
<code>USBH_FUNCTION</code>	Is used as a member for the <code>USBH_HEADER</code> data structure.
<code>USBH_POWER_STATE</code>	Specifies some power states.

**Table 4.21: emUSB-Host data structure overview**

## 4.3.1 USBH\_INTERFACE\_MASK

### Definition

```
typedef struct tag_USBH_INTERFACE_MASK {
    U16 Mask;
    U16 VID;
    U16 PID;
    U16 bcdDevice;
    U8 Interface;
    U8 Class;
    U8 SubClass;
    U8 Protocol;
} USBH_INTERFACE_MASK;
```

### Description

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

### Members

Member	Description
<a href="#">Mask</a>	Contains an or'ed selection of the following flags. If the flag is set the related member of this structure is compared to the properties of the USB interface.  USBH_INFO_MASK_VID Compare the vendor ID (VID) of the device. USBH_INFO_MASK_PID Compare the product ID (PID) of the device. USBH_INFO_MASK_DEVICE Compare the bcdDevice value of the device. USBH_INFO_MASK_INTERFACE Compare the interface number. USBH_INFO_MASK_CLASS Compare the class of the interface. USBH_INFO_MASK_SUBCLASS Compare the sub class of the interface. USBH_INFO_MASK_PROTOCOL Compare the protocol of the interface.
<a href="#">VID</a>	Contains a vendor ID.
<a href="#">PID</a>	Contains a product ID.
<a href="#">bcdDevice</a>	Contains a BCD coded device version.
<a href="#">Interface</a>	Contains the interface number.
<a href="#">Class</a>	Describes the class code stored in the interface.
<a href="#">Subclass</a>	Describes the sub class code stored in the interface.
<a href="#">Protocol</a>	Describes the protocol stored in the interface.

**Table 4.22: USBH\_INTERFACE\_MASK() member list**

## 4.3.2 USBH\_INTERFACE\_INFO

### Definition

```
typedef struct tag_USBH_INTERFACE_INFO {
    USBH_INTERFACE_ID InterfaceID;
    USB_DEVICE_ID      DeviceID;
    U16                VID;
    U16                PID;
    U16                bcdDevice;
    U8                 Interface;
    U8                 Class;
    U8                 SubClass;
    U8                 Protocol;
    unsigned int       OpenCount;
    U8                 ExclusiveUsed;
    USB_SPEED          Speed;
    U8                 SerialNumber[256];
    U8                 SerialNumberSize;
} USBH_INTERFACE_INFO;
```

### Description

Is used to get information about a device with the function `USBH_GetInterfaceInfo`.

### Members

Member	Description
<code>InterfaceID</code>	Contains the unique interface ID. This ID is assigned if the USB device was successful enumerated. It is valid until the device is removed for the host. If the device is reconnected a different interface ID is assigned to each interface.
<code>DeviceID</code>	Contains the unique device 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 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.
<code>VID</code>	Contains the vendor ID.
<code>PID</code>	Contains the product ID.
<code>bcdDevice</code>	Contains the BCD coded device version.
<code>Interface</code>	Contains the USB interface number.
<code>Class</code>	Specifies the interface class.
<code>Subclass</code>	Specifies the interface sub class.
<code>Protocol</code>	Specifies the interface protocol.
<code>OpenCount</code>	Specifies the number of open handles for this interface.
<code>ExclusiveUsed</code>	Determines if this interface is used exclusive.
<code>Speed</code>	Specifies the operation speed of this interface.
<code>SerialNumber[256]</code>	Contains the serial number as a counted UNICODE string.
<code>SerialNumberSize</code>	Contains the length of the serial number in bytes.

**Table 4.23: USBH\_INTERFACE\_INFO() member list**

### 4.3.3 USBH\_ENUM\_ERROR

#### Definition

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

#### Description

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

#### Members

Member	Description
Flags	<p>Additional flags to determine the location and the type of the error.</p> <p><code>USBH_ENUM_ERROR_EXTHUBPORT_FLAG</code> means the device is connected to an external hub.</p> <p><code>USBH_ENUM_ERROR_RETRY_FLAG</code> the bus driver retries the enumeration of this device automatically.</p> <p><code>USBH_ENUM_ERROR_STOP_ENUM_FLAG</code> the bus driver does not restart the enumeration for this device because all retries has failed. The application can force the bus driver to restart the enumeration by calling the function <code>USBH_RestartEnumError</code>.</p> <p><code>USBH_ENUM_ERROR_DISCONNECT_FLAG</code> 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 <code>USBH_RestartEnumError</code> cannot re-enumerate the device.</p> <p><code>USBH_ENUM_ERROR_ROOT_PORT_RESET</code> means an error during the USB reset of a root hub port occurs.</p> <p><code>USBH_ENUM_ERROR_HUB_PORT_RESET</code> means an error during a reset of an external hub port occurs.</p> <p><code>UDB_ENUM_ERROR_INIT_DEVICE</code> means an error during the device initialization (e.g. no answer to a descriptor request or it failed other standard requests).</p> <p><code>UDB_ENUM_ERROR_INIT_HUB</code> means the enumeration of an external hub fails.</p>

**Table 4.24: USBH\_ENUM\_ERROR() member list**

Member	Description
<a href="#">PortNumber</a>	Port number of the parent port where the USB device is connected. A flag in the PortFlags field determine if this is an external hub port.
<a href="#">Status</a>	Status of the failed operation.
<a href="#">ExtendedErrorInformation</a>	Internal information used for debugging.

**Table 4.24: USBH\_ENUM\_ERROR() member list**

## 4.3.4 USBH\_EP\_MASK

### Definition

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

### Description

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

### Members

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

**Table 4.25: USBH\_EP\_MASK() member list**

## 4.3.5 USBH\_CONTROL\_REQUEST

### Definition

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

### Description

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

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

SetAddress

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

Clear Feature Endpoint Halt

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

SetConfiguration

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

### Members

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

**Table 4.26: USBH\_CONTROL\_REQUEST() member list**

## 4.3.6 USBH\_BULK\_INT\_REQUEST

### Definition

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

### Description

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

### Members

Member	Description
<a href="#">Endpoint</a>	Specifies the endpoint address with direction bit.
<a href="#">Buffer</a>	Pointer to a caller provided buffer.
<a href="#">Length</a>	Contains the size of the buffer and returns the number of bytes transferred.

**Table 4.27: USBH\_BULK\_INT\_REQUEST() member list**

## 4.3.7 USBH\_ISO\_FRAME

### Definition

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

### Description

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

### Members

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

**Table 4.28: USBH\_ISO\_FRAME() member list**

## 4.3.8 USBH\_ISO\_REQUEST

### Definition

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

### Description

Is incomplete defined. That means the data structure consists of this data structure and an array of data structures `USBH_ISO_FRAME`. The size of the array is defined by `Frames`. Use the macro `USBH_GET_ISO_URB_SIZE` to get the size for an ISO URB.

### Members

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

**Table 4.29: USBH\_ISO\_REQUEST() member list**

## 4.3.9 USBH\_ENDPOINT\_REQUEST

### Definition

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

### Description

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

### Members

Member	Description
<a href="#">Endpoint</a>	Specifies the endpoint address.

**Table 4.30: USBH\_ENDPOINT\_REQUEST() member list**

## 4.3.10 USBH\_SET\_CONFIGURATION

### Definition

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

### Description

is used with the request `USBH_FUNCTION_SET_CONFIGURATION`.

### Members

Member	Description
<code>ConfigurationDescriptorIndex</code>	Specifies the index in the configuration description.

**Table 4.31: USBH\_SET\_CONFIGURATION() member list**

## 4.3.11 USBH\_SET\_INTERFACE

### Definition

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

### Description

is used with the request `USBH_FUNCTION_SET_INTERFACE`.

### Members

Member	Description
<a href="#">AlternateSetting</a>	Specifies the alternate setting.

**Table 4.32: USBH\_SET\_INTERFACE() member list**

## 4.3.12 USBH\_SET\_POWER\_STATE

### Definition

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

### Description

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

### Members

Member	Description
<a href="#">PowerState</a>	Specifies the power state

**Table 4.33: USBH\_SET\_POWER\_STATE() member list**

## 4.3.13 URB

### Definition

```
typedef struct tag_URB {
    USBH_HEADER Header;
    union
        Request;
} URB;
```

### Description

The following table lists the possible information types and associated structures:

Request Type	Associated Structure
<a href="#">ControlRequest</a>	USBH_CONTROL_REQUEST
<a href="#">BulkIntRequest</a>	USBH_BULK_INT_REQUEST
<a href="#">IsoRequest</a>	USBH_ISO_REQUEST
<a href="#">EndpointRequest</a>	USBH_ENDPOINT_REQUEST
<a href="#">SetConfiguration</a>	USBH_SET_CONFIGURATION
<a href="#">SetInterface</a>	USBH_SET_INTERFACE
<a href="#">SetPowerState</a>	USBH_SET_POWER_STATE

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

### Members

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

**Table 4.34: URB() member list**

## 4.3.14 USBH\_PNP\_NOTIFICATION

### Definition

```
typedef struct tag_USBH_PNP_NOTIFICATION {
    USBH_PnpNotification * PnpNotification;
    void * Context;
    USBH_INTERFACE_MASK InterfaceMask;
} USBH_PNP_NOTIFICATION;
```

### Description

Is used as an input parameter for the `USBH_RegisterPnPNotification` function.

### Members

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

**Table 4.35: USBH\_PNP\_NOTIFICATION() member list**

## 4.3.15 USBH\_HEADER

### Definition

```
typedef struct tag_USBH_HEADER {
    USBH_FUNCTION           Function;
    USBH_STATUS             Status;
    USBH_ON_COMPLETION_FUNC * Completion;
    void                    * Context;
    DLIST                   ListEntry;
} USBH_HEADER;
```

### Description

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

### Members

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

**Table 4.36: USBH\_HEADER() member list**

## 4.3.16 USBH\_SPEED

### Definition

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

### Description

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

### Members

Member	Description
<a href="#">USBH_SPEED_UNKNOWN</a>	The speed is unknown.
<a href="#">USBH_LOW_SPEED</a>	The device operates at low speed.
<a href="#">USBH_FULL_SPEED</a>	The device operates at full speed.
<a href="#">USBH_HIGH_SPEED</a>	The device operates at high speed.

**Table 4.37: USBH\_SPEED() member list**

## 4.3.17 USBH\_PNP\_EVENT

### Definition

```
typedef enum tag_USBH_PNP_EVENT {  
    USBH_AddDevice,  
    USBH_RemoveDevice  
} USBH_PNP_EVENT;
```

### Description

Is used as a parameter for the PnP notification.

### Members

Member	Description
<a href="#">USBH_AddDevice</a>	Indicates that a device was connected to the host and new interface is available.
<a href="#">USBH_RemoveDevice</a>	Indicates that a device has been removed.

**Table 4.38: USBH\_PNP\_EVENT() member list**

## 4.3.18 USBH\_FUNCTION

### Definition

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

### Description

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

### Entries

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

Table 4.39: `USBH_FUNCTION()` member list

Entry	Description
<p><code>USBH_FUNCTION_RESET_DEVICE</code></p>	<p>Sends an USB reset to the device. This causes a remove event for all interfaces of the device. After the device is successfully enumerated an arrival event is indicated. All interfaces get new interface ID's. This request uses only the URB header. If the driver indicates an device arrival event the device is in a defined state because it is reseted and enumerated by the bus driver. This request can be part of an error recovery or part of special class protocols like DFU. The application should abort all pending requests and close all handles to this device. All handles become invalid.</p>
<p><code>USBH_FUNCTION_RESET_ENDPOINT</code></p>	<p>Clears an error condition on a special endpoint. If a data transfer error occurs that cannot be handled in hardware the bus driver stops the endpoint and does not allow further data transfers before the endpoint is reseted 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 <code>USBH_FUNCTION_ABORT_ENDPOINT</code>. This function uses the data structure <code>USBH_ENDPOINT_REQUEST</code>.</p>
<p><code>USBH_FUNCTION_ABORT_ENDPOINT</code></p>	<p>Aborts all pending requests on a endpoint. The host controller calls the completion function with a status code <code>USBH_STATUS_CANCELED</code>. The completion of the URBs may be delayed. The application should wait until all pending requests has been returned by the driver before the handle is closed or <code>USBH_FUNCTION_RESET_ENDPOINT</code> is called.</p>

**Table 4.39: USBH\_FUNCTION() member list**

Entry	Description
<p><code>USBH_FUNCTION_SET_CONFIGURATION</code></p>	<p>The driver selects the configuration defined by the configuration descriptor with the index 0 during the enumeration. If the application uses this configuration there is no need to call this function. If the application wants to activate a different configuration this function must be called.</p>
<p><code>USBH_FUNCTION_SET_INTERFACE</code></p>	<p>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 <code>USBH_SET_INTERFACE</code>.</p>
<p><code>USBH_FUNCTION_SET_POWER_STATE</code></p>	<p>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 <code>USBH_SET_POWER_STATE</code>. After the enumeration the device is in normal power state.</p>

**Table 4.39: USBH\_FUNCTION() member list**

## 4.3.19 USBH\_POWER\_STATE

### Definition

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

### Description

Is used as a member in the USBH\_SET\_POWER\_STATE data structure.

### Members

Member	Description
<a href="#">USBH_NORMAL_POWER</a>	The device is switched to normal operation.
<a href="#">USBH_SUSPEND</a>	The device is switched to USB Suspend mode.

**Table 4.40: USBH\_POWER\_STATE() member list**

## 4.4 Function Types

The table below lists the available function types.

Structure	Description
USBH_ON_PNP_EVENT_FUNC	Is called by the library if a PnP event occurs and if a PnP notification was registered.
USBH_ON_ENUM_ERROR_FUNC	Contains information about a USB interface and the related device.
USBH_ON_COMPLETION_FUNC	Is used as an notification parameter for the USBH_EnumErrorNotification function.

**Table 4.41: emUSB-Host function type overview**

## 4.4.1 USBH\_ON\_PNP\_EVENT\_FUNC

### Definition

```
typedef void USBH_ON_PNP_EVENT_FUNC (
    void          * Context,
    USBH_PNP_EVENT Event,
    USBH_INTERFACE_ID InterfaceID );
```

### Description

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

### Parameters

Parameter	Description
<a href="#">Context</a>	Is the user defined pointer that was passed to <code>USBH_RegisterPnPNotification</code> . The library does not modify this parameter.
<a href="#">Event</a>	Specifies the PnP event.
<a href="#">InterfaceID</a>	Contains the interface ID of the removed or added interface.

**Table 4.42: USBH\_ON\_PNP\_EVENT\_FUNC() parameter list**

## 4.4.2 USBH\_ON\_ENUM\_ERROR\_FUNC

### Definition

```
typedef void USBH_ON_ENUM_ERROR_FUNC(
    void * Context,
    const USBH_ENUM_ERROR * EnumError );
```

### Description

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

### Parameters

Parameter	Description
Context	Is a user defined pointer that was passed to USBH_RegisterEnumErrorNotification.
EnumError	Specifies the enumeration error. This pointer is temporary and must not be access after the functions returns.

**Table 4.43: USBH\_ON\_ENUM\_ERROR\_FUNC() parameter list**

### 4.4.3 USBH\_ON\_COMPLETION\_FUNC

#### Definition

```
typedef void USBH_ON_COMPLETION_FUNC ( tag_URB * Urb );
```

#### Description

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

#### Parameter

Parameter	Description
<a href="#">Urb</a>	Contains the URB that was completed.

**Table 4.44: USBH\_ON\_COMPLETION\_FUNC() parameter list**

## 4.5 Use of undocumented functions

Functions, variables and data-types which 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.



# Chapter 5

## Mass Storage Device

---

This chapter describes the emUSB-Host Mass storage device class driver and its usage.

## 5.1 Introduction

The USB Host MSD class software allows accessing USB Mass Storage Devices.

It implements the USB Mass Storage Device class protocols specified by the USB Implementers Forum. The entire API of this class driver is prefixed USBH\_MSD\_.

This chapter describes the architecture, the features and the programming interface of the code.

### 5.1.1 Overview

A mass storage device connected to the USB host is added to the file system as device. All operations on the device, such as formatting, reading / writing of files and directories are performed through the API of the file system. With emFile, the device name of the first MSD is "msd:0:".

### 5.1.2 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.

### 5.1.3 Restrictions

The following restrictions relate to the USB Host library:

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

### 5.1.4 Requirements

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

### 5.1.5 Example code

Example code which is provided in the file OS\_USBH\_MSD.c.

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

### 5.1.6 Supported Protocols

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

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

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

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

## 5.2 API Functions

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

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

**Table 5.1: emUSB-Host MSD API function overview**

## 5.2.1 USBH\_MSD\_Init()

### Description

Initializes the USBH MSD library.

### Prototype

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

### Parameters

Parameter	Description
<code>pLunNotification</code>	Pointer to a function that shall be called when a new device notification is received. This means when a device is attached and ready or when it is removed.
<code>pContext</code>	Pointer to a context that should be passed when the <code>pLunNotification</code> is called.

**Table 5.2: USBH\_MSD\_Init() parameter list**

### Additional information

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

### Example:

```

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

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

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

```

## 5.2.2 USBH\_MSD\_Exit()

### Description

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

### Prototype

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

### Additional information

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

## 5.2.3 USBH\_MSD\_ReadSectors()

### Description

Reads sectors from a USB Mass Storage device.

### Prototype

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

### Parameters

Parameter	Description
<code>UnitId</code>	0-based Unit Id. The first unit in the system has UnitID of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIDs at run time.
<code>SectorIndex</code>	0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, ...
<code>NumSectors</code>	Determines the number of sectors to read.
<code>pBuffer</code>	Pointer to a byte buffer. The caller is responsible for the storage of the buffer.

**Table 5.3: USBH\_MSD\_ReadSectors() parameter list**

### Return Value

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

## 5.2.4 USBH\_MSD\_WriteSectors()

### Description

Writes sectors to a USB Mass Storage device.

### Prototype

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

### Parameters

Parameter	Description
<code>UnitId</code>	0-based Unit Id. The first unit in the system has UnitID of 0, the second one a value of 1. If you are dealing with multiple devices or devices with multiple LUNs, it is good practice to retrieve the UnitIDs at run time.
<code>SectorIndex</code>	0-based sector index: of the first sector to read. First sector has index 0, second sector has index 1, ...
<code>NumSectors</code>	Determines the number of sectors to write.
<code>pBuffer</code>	Pointer to a buffer containing the data to be written.

**Table 5.4: USBH\_MSD\_WriteSectors() parameter list**

### Return Value

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

## 5.2.5 USBH\_MSD\_GetUnitInfo()

### Description

Returns basic information about the logical unit (LUN).

### Prototype

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

### Parameters

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

**Table 5.5: USBH\_MSD\_GetUnitInfo() parameter list**

### Return Value

Returns `USBH_MSD_STATUS_SUCCESS` in case of success. If the device is not a USB Mass Storage device, `USBH_MSD_STATUS_ERROR` will be returned. `USBH_MSD_STATUS_TIMEOUT` is returned if the function call timed out.

## 5.2.6 USBH\_MSD\_GetStatus()

### Description

Checks the state of a device unit.

### Prototype

```
int USBH_MSD_GetStatus( U8 UnitId );
```

### Parameter

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

**Table 5.6: USBH\_MSD\_GetStatus() parameter list**

### Return Value

If the device is working, `USBH_MSD_STATUS_SUCCESS` is returned. If the device does not work correctly or is disconnected the function returns `USBH_MSD_STATUS_ERROR`.

## 5.3 Data Structures

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

Structure	Description
<a href="#">USBH_MSD_UNIT_INFO</a>	Contains logical unit information.

**Table 5.7: emUSB-Host MSD data structure overview**

### 5.3.1 USBH\_MSD\_UNIT\_INFO

#### Definition

```
typedef struct tag_USB_MSD_UNIT_INFO {  
    U32 TotalSectors;  
    U16 BytesPerSector;  
} USBH_MSD_UNIT_INFO;
```

#### Description

Contains logical unit information.

#### Parameters

Parameter	Description
TotalSectors	Contains the number of total sectors available on the LUN.
BytesPerSector	Contains the number of bytes per sector.

**Table 5.8: USBH\_MSD\_UNIT\_INFO() parameter list**

## 5.4 Function Types

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

Structure	Description
<a href="#">USBH_MSD_LUN_NOTIFICATION_FUNC</a>	Type of callback set in USBH_MSD_Init.

**Table 5.9: emUSB-Host MSD function type overview**

## 5.4.1 USBH\_MSD\_LUN\_NOTIFICATION\_FUNC

### Definition

```
typedef void tag_USB_MSD_LUN_NOTIFICATION_FUNC {
    void          pContext;
    U8            DevIndex;
    USBH_MSD_EVENT Event
} USBH_MSD_UNIT_INFO;
```

### Description

This callback function is called when new a logical units is added or removed. To get detailed information USBH\_MSD\_GetLuns has to be called. The LUN indexes must be used to get access to an specified unit of the device.

### Parameters

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

**Table 5.10: USBH\_MSD\_LUN\_NOTIFICATION\_FUNC() parameter list**

## 5.5 Error Codes

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

Error Code	Description
USBH_MSD_STATUS_SUCCESS	(0)
USBH_MSD_STATUS_ERROR	(-1)
USBH_MSD_STATUS_PARAMETER	(-2)
USBH_MSD_STATUS_LENGTH	(-3)
USBH_MSD_STATUS_TIMEOUT	(-4)
USBH_MSD_STATUS_COMMAND_FAILED	(-5)
USBH_MSD_STATUS_INTERFACE_PROTOCOL	(-6)
USBH_MSD_STATUS_INTERFACE_SUB_CLASS	(-7)
USBH_MSD_STATUS_PIPE_STALLED	(-9)
USBH_MSD_STATUS_TRANSMISSION	(-10)
USBH_MSD_STATUS_SENSE_STOP	(-11)
USBH_MSD_STATUS_SENSE_REPEAT	(-12)
USBH_MSD_STATUS_WRITE_PROTECT	(-13)

**Table 5.11: emUSB-Host MSD error code overview**

### 5.5.1 USBH\_MSD\_STATUS\_SUCCESS

#### Description

The operation has been successfully completed.

### 5.5.2 USBH\_MSD\_STATUS\_ERROR

#### Description

The operation has been completed with an error.

### 5.5.3 USBH\_MSD\_STATUS\_PARAMETER

#### Description

A parameter is incorrect.

### 5.5.4 USBH\_MSD\_STATUS\_LENGTH

#### Description

The operation detected a length error.

### 5.5.5 USBH\_MSD\_STATUS\_TIMEOUT

#### Description

The timeout of the operation has expired. This error code is used in all layers.

### 5.5.6 USBH\_MSD\_STATUS\_COMMAND\_FAILED

#### Description

This error is reported if the command code was sent successfully but the status returned from the device indicates a command error.

### 5.5.7 USBH\_MSD\_STATUS\_INTERFACE\_PROTOCOL

#### Description

The used interface protocol is not supported. The interface protocol is defined by the interface descriptor.

### 5.5.8 USBH\_MSD\_STATUS\_INTERFACE\_SUB\_CLASS

#### Description

The used interface sub class is not supported. The interface sub class is defined by the interface descriptor.

### 5.5.9 USBH\_MSD\_STATUS\_PIPE\_STALLED

#### Description

A pipe is stalled. This error is reported from the USB driver layer.

### 5.5.10 USBH\_MSD\_STATUS\_TRANSMISSION

#### Description

A USB bus error occurred. This may be caused by a CRC error, a toggle error or another USB bus error. This error is reported from the USB driver layer.

### **5.5.11 USBH\_MSD\_STATUS\_SENSE\_STOP**

#### **Description**

This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library will not repeat the command.

### **5.5.12 USBH\_MSD\_STATUS\_SENSE\_REPEAT**

#### **Description**

This error indicates that the device has not accepted the command. The execution result of the command is stored in the sense element of the unit. The library repeats the command after detection of the sense code.

### **5.5.13 USBH\_MSD\_STATUS\_WRITE\_PROTECT**

#### **Description**

This error indicates that the medium is write protected. It will be returned by `USBH_MSD_WriteSectors` if writing to the medium is not allowed.



# Chapter 6

## Human Interface Device

---

This chapter describes the emUSB-Host Human interface device class driver and its usage.

## 6.1 Introduction

The USB 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 `USBH_HID_`.

This chapter describes the architecture, the features and the programming interface of the code.

### 6.1.1 Overview

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

### 6.1.2 Restrictions

The following restrictions relate to the USB Host library:

- Currently, only Keyboard and mouse devices are supported. The HID report descriptor is not currently parsed.

### 6.1.3 Example code

Example code which is provided in the file `OS_USBH_HID.c`.

The example outputs mouse and keyboard events to the terminal (using `printf`).

## 6.2 API Functions

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

Function	Description
<code>USBH_HID_Init()</code>	Initializes the USBH HID library.
<code>USBH_HID_Exit()</code>	Releases all resources, closes all handles to the USB bus driver and unregisters all notification functions.
<code>USBH_HID_SetOnMouseStateChange()</code>	Set function to be called in case of mouse events.
<code>USBH_HID_SetOnKeyboardStateChange()</code>	Set function to be called in case of keyboard events.

**Table 6.1: emUSB-Host HID API function overview**

## 6.2.1 USBH\_HID\_Init()

### Description

Initializes the USBH HID library.

### Prototype

```
int USBH_HID_Init();
```

### Additional information

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

## 6.2.2 USBH\_HID\_Exit()

### Description

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

### Prototype

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

### Additional information

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

## 6.2.3 USBH\_HID\_SetOnMouseStateChange()

### Description

Set function to be called in case of mouse events.

### Prototype

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

### Example

```
/******  
*  
*      _OnMouseChange  
*/  
static void _OnMouseChange(USBH_HID_MOUSE_DATA * pMouseData) {  
    _MouseData      = *pMouseData;  
    _EventOccurred |= MOUSE_EVENT;  
    OS_EVENT_Pulse(&_Event);  
}
```

## 6.2.4 USBH\_HID\_SetOnKeyboardStateChange()

### Description

Set function to be called in case of keyboard events.

### Prototype

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

### Example

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

## 6.3 Data Structures

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

Structure	Description
<a href="#">USBH_HID_KEYBOARD_DATA</a>	Contains keyboard state information.
<a href="#">USBH_HID_MOUSE_DATA</a>	Contains mouse state information.

**Table 6.2: emUSB-Host HID data structure overview**

## 6.3.1 USBH\_HID\_KEYBOARD\_DATA

### Definition

```
typedef struct {
    unsigned code;
    int value;
} USBH_HID_KEYBOARD_DATA;
```

### Description

Contains keyboard state information.

### Parameters

Parameter	Description
<code>code</code>	Contains the keycode.
<code>value</code>	Keyboard state info. Refer to sample code for more information.

**Table 6.3: USBH\_HID\_KEYBOARD\_DATA() parameter list**

## 6.3.2 USBH\_HID\_MOUSE\_DATA

### Definition

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

### Description

Contains mouse state information.

### Parameters

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

**Table 6.4: USBH\_HID\_MOUSE\_DATA() parameter list**

## 6.4 Function Types

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

Structure	Description
<a href="#">USBH_HID_ON_KEYBOARD_FUNC</a>	This callback function is called when a key is pressed or released.
<a href="#">USBH_HID_ON_MOUSE_FUNC</a>	This callback function is called when the mouse is moved or a button is pressed or released.

**Table 6.5: emUSB-Host HID function type overview**

## 6.4.1 USBH\_HID\_ON\_KEYBOARD\_FUNC

### Definition

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

### Description

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

### Parameter

Parameter	Description
<a href="#">pKeyData</a>	Points to the structure containing status information.

**Table 6.6: USBH\_HID\_ON\_KEYBOARD\_FUNC() parameter list**

## 6.4.2 USBH\_HID\_ON\_MOUSE\_FUNC

### Definition

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

### Description

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

### Parameter

Parameter	Description
<a href="#">pMouseData</a>	Points to the structure containing status information.

**Table 6.7: USBH\_HID\_ON\_MOUSE\_FUNC() parameter list**



# Chapter 7

## Configuring emUSB-Host

---

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

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

## 7.1 Runtime configuration

Every driver folder includes a configuration file with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

### 7.1.1 Driver handling

`USBH_X_Config()` is called at initialization of the USB Host stack. It is called by the USB Host stack during `USBH_Init()`. `USBH_X_Config()` should help to bundle the process of adding and configuring the driver.

#### 7.1.1.1 `USBH_X_Config()`

##### Description

Helper function to prepare and configure the USB Host stack.

##### Prototype

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

##### Additional information

This function is called by the startup code of the USB Host stack from `USBH_Init()`.

## 7.2 Compile-time configuration

The following types of configuration macros exist:

### Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

### Numerical values "N"

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

### Function replacements "F"

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

### 7.2.1 Compile-time configuration switches

Type	Symbolic name	Default	Description
Debug macros			
N	<a href="#">USBH_DEBUG</a>	0	Macro to define the debug level of the emUSB-Host build.
Optimization macros			
F	<a href="#">USBH_MEMCPY</a>	memcpy (C-routine in standard C-library)	Macro to define an optimized memcpy routine to speed up the stack. An optimized memcpy routine is typically implemented in assembly language. Optimized version for the IAR compiler is supplied.
F	<a href="#">USBH_MEMSET</a>	memset (C-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	<a href="#">USBH_MEMMOVE</a>	memmove (C-routine in standard C-library)	Macro to define an optimized memmove routine to speed up the stack. An optimized memmove routine is typically implemented in assembly language.
F	<a href="#">USBH_MEMCMP</a>	memcmp (C-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.

### 7.2.2 Debug level

emUSB-Host can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, emUSB-Host uses the logging routines. These routines can be blank, they are not required for the

functionality of emUSB-Host. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

If (USBH\_DEBUG == 0):  
used for release builds. Includes no debug options.

If (USBH\_DEBUG == 1):  
USBH\_PANIC() is mapped to USBH\_Panic().

If (USBH\_DEBUG >= 2):  
USBH\_PANIC() is mapped to USBH\_Panic() and logging support is activated.

# Chapter 8

## Debugging

---

emUSB-Host comes with various debugging options. These includes optional warning and log outputs, as well as other run-time options which perform checks at run time as well as options to drop incoming or outgoing packets to test stability of the implementation on the target system.

## 8.1 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 USB 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.

## 8.2 API functions

Function	Description
Filter functions	
<code>USBH_SetLogFilter()</code>	Sets the mask that defines which logging message should be displayed.
<code>USBH_SetWarnFilter()</code>	Sets the mask that defines which warning message should be displayed.
<code>USBH_AddLogFilter()</code>	Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.
<code>USBH_AddWarnFilter()</code>	Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.
General debug functions/macros	
<code>USBH_LOG()</code>	Called if the stack encounters a critical situation.
<code>USBH_WARN()</code>	Called if the stack encounters a critical situation.
<code>USBH_PANIC()</code>	Called if the stack encounters a critical situation.

**Table 8.1: emUSB-Host debugging API function overview**

## 8.2.1 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 );
```

### Parameter

Parameter	Description
<code>FilterMask</code>	Specifies which logging messages should be displayed.

**Table 8.2: USBH\_SetLogFilter() parameter list**

### Additional information

Should be called from `USBH_X_Config()`. By default, the filter condition `USBH_MTYPE_INIT` is set.

## 8.2.2 USBH\_SetWarnFilter()

### Description

Sets a mask that defines which warning messages should be logged. Warning messages are only available in debug builds of emUSB-Host.

### Prototype

```
void USBH_SetWarnFilter( U32 FilterMask );
```

### Parameter

Parameter	Description
<a href="#">FilterMask</a>	Specifies which warning messages should be displayed.

**Table 8.3: USBH\_SetWarnFilter() parameter list**

### Additional information

Should be called from `USBH_X_Config()`. By default, all filter conditions are set.

## 8.2.3 USBH\_AddLogFilter()

### Description

Adds an additional filter condition to the mask which specifies the logging messages that should be displayed.

### Prototype

```
void USBH_AddLogFilter( U32 FilterMask );
```

### Parameter

Parameter	Description
<code>FilterMask</code>	Specifies which logging messages should be added to the filter mask.

**Table 8.4: USBH\_AddLogFilter() parameter list**

### Additional information

`USBH_AddLogFilter()` 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.

## 8.2.4 USBH\_AddWarnFilter()

### Description

Adds an additional filter condition to the mask which specifies the warning messages that should be displayed.

### Prototype

```
void USBH_AddWarnFilter( U32 FilterMask );
```

### Parameter

Parameter	Description
<code>FilterMask</code>	Specifies which warning messages should be added to the filter mask.

**Table 8.5: USBH\_AddWarnFilter() parameter list**

### Additional information

`USBH_AddWarnFilter()` 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.

## 8.2.5 USBH\_LOG()

### Description

This macro maps to a function in debug builds only. The function outputs logging messages. In a release build, this macro is defined empty.

### Prototype

```
USBH_LOG( const char * s );
```

### Parameter

Parameter	Description
<a href="#">s</a>	Pointer to the string to be sent.

**Table 8.6: USBH\_LOG() parameter list**

## 8.2.6 USBH\_WARN()

### Description

This macro maps to a function in debug builds only. The function outputs warning messages. In a release build, this macro is defined empty.

### Prototype

```
USBH_WARN( const char * s );
```

### Parameter

Parameter	Description
<a href="#">s</a>	Pointer to the string to be sent.

**Table 8.7: USBH\_WARN() parameter list**

## 8.2.7 USBH\_PANIC()

### Description

This macro is called by the stack code when it detects a situation that should not be occurring and the stack can not continue. The intention for the `USBH_PANIC()` macro is to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint.

### Prototype

```
USBH_PANIC ( const char * sError );
```

### Parameter

Parameter	Description
<code>sError</code>	Pointer to the string to be sent.

**Table 8.8: USBH\_PANIC() parameter list**

### Additional information

This macro maps to a function in debug builds only. If `USBH_DEBUG > 0`, the macro maps to the stack internal function `void USBH_Panic (const char * sError)`. `USBH_Panic()` disables all interrupts to avoid further task switches, outputs `sError` via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error code is passed to the function as parameter.

In a release build, this macro is defined empty, so that no additional code will be included by the linker.

## 8.3 Message types

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings.

Symbolic name	Description
USBH_MTYPE_INIT	Activates output of messages from the initialization of the stack that should be logged.
USBH_MTYPE_CORE	Activates output of messages from the core of the stack that should be logged.
USBH_MTYPE_ALLOC	Activates output of messages from the memory allocating module of the stack that should be logged.
USBH_MTYPE_DRIVER	Activates output of messages from the driver that should be logged.
USBH_MTYPE_MEM	Activates output of messages from the memory that should be logged.
USBH_MTYPE_OHCI	Activates output of messages from the Open Host Controller Interface that should be logged.
USBH_MTYPE_UBD	
USBH_MTYPE_PNP	
USBH_MTYPE_DEVICE	
USBH_MTYPE_HUB	
USBH_MTYPE_MSD	
USBH_MTYPE_HID	
USBH_MTYPE_APPLICATION	

**Table 8.9: USB Host message types**



# Chapter 9

## OS integration

---

emUSB-Host is designed to be used in a multitasking environment. The interface to the operating system is encapsulated in a single file, the USB-Host/OS interface. For emUSB-Host, all functions required for this USB-Host/OS interface are implemented in a single file which comes with emUSB-Host.

This chapter provides descriptions of the functions required to fully support emUSB-Host in multitasking environments.

## 9.1 General information

The complexity of the IP/OS Interface depends on the task model selected. All OS interface functions for emUSB-Host are implemented in `USBH_OS.c` which is located in the root folder of the USB Host stack.

## 9.2 OS layer API functions

Function	Description
General macros	
<code>USBH_OS_Delay()</code>	Blocks the calling task for a given time.
<code>USBH_OS_DisableInterrupt()</code>	Disables interrupts.
<code>USBH_OS_EnableInterrupt()</code>	Enables interrupts.
<code>USBH_OS_GetTime32()</code>	Returns the current system time in ticks. Return the current system time in ms. On 32-bit systems, the value will wrap around after approximately 49.7 days. This is taken into account by the stack.
<code>USBH_OS_Init()</code>	Creates and initializes all objects required for task synchronization. These are 2 events (for <code>USBH_Task</code> and <code>USBH_RxTask</code> ) and one semaphore for protection of critical code which may not be executed from multiple task at the same time.
<code>USBH_OS_Lock()</code>	The stack requires a single lock, typically a resource semaphore or mutex. This function locks this object, guarding sections of the stack code against other tasks. If the entire stack executes from a single task, no functionality is required here.
<code>USBH_OS_Unlock()</code>	Unlocks the single lock used locked by a previous call to <code>USBH_OS_Lock()</code> .
USBH_Task synchronization	
<code>USBH_OS_SignalNetEvent()</code>	Wakes the <code>USBH_Task</code> if it is waiting for a NET-event or timeout in the function <code>USBH_OS_WaitNetEvent()</code> .
<code>USBH_OS_WaitNetEvent()</code>	Called from <code>USBH_Task</code> only. Blocks until the timeout expires or a NET-event occurs, meaning <code>USBH_OS_SignalNetEvent()</code> is called from an other task or ISR.
USBH_RxTask synchronization	
<code>USBH_OS_SignalRxEvent()</code>	Wakes the <code>USBH_RxTask</code> if it is waiting for a NET-event or timeout in the function <code>USBH_OS_WaitRxEvent()</code> .
<code>USBH_OS_WaitRxEvent()</code>	Optional. Called from <code>USBH_RxTask</code> , if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning <code>USBH_OS_SignalRxEvent()</code> is called from the ISR.
Application task synchronization	
<code>USBH_OS_WaitItem()</code>	Suspend a task which needs to wait for a object. This object is identified by a pointer to it and can be of any type, for example a socket.
<code>USBH_OS_WaitItemTimed()</code>	Suspend a task which needs to wait for a object. This object is identified by a pointer to it and can be of any type, for example a socket. The second parameter defines the maximum time in timer ticks until the event have to be signaled.
<code>USBH_OS_SignalItem()</code>	Sets an event object to signaled state, or resumes tasks which are waiting at the event object. Function is called from a task, not an ISR.

**Table 9.1: Target OS interface function list**

### 9.2.1 Examples

#### OS interface routine for emUSB-Host

All OS interface routines are implemented in `USBH_OS.c` which is located in the root folder of the USB Host stack.

# Chapter 10

## 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.

## 10.1 Memory footprint

emUSB-Host is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. Note that the values are only valid for the given configuration.

### System

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	ARM7
Tool chain	IAR Embedded Workbench for Cortex-M3, V5.30
Compiler options	Highest size optimization;

**Table 10.1: ARM7 sample configuration**

### 10.1.1 ROM

The following table shows the ROM requirement of emUSB-Host:

Description	ROM
emUSB-Host core incl. driver	app. 20 KBytes
HID class support	app. 5 KBytes
MSD class support	app. 8 KBytes + sizeof(Filesystem)*

The memory requirements of a interface driver is about 1.5 - 2.0 Kbytes.

### 10.1.2 RAM

The following table shows the ROM requirement of emUSB-Host:

Description	ROM
emUSB-Host core incl. driver	app. 20 Kbytes

The memory requirements of a interface driver is about 1.5 - 2.0 Kbytes.

\* ROM size of emFile File system is app. 10KBytes

## 10.2 Performance

### System

Detail	Description
CPU	ARM7 with integrated MAC running with 48Mhz
Tool chain	IAR Embedded Workbench for Cortex-M3 V530
Compiler options	Highest speed optimization;

**Table 10.2: ARM7 sample configuration**

The following table shows the send and receive speed of emUSB-Host:

Description	Speed
<b>Bulk</b>	
Send speed	400-1000 KByte/sec
Receive speed	400-1000 KByte/sec



# Chapter 11

## Related Documents

---

- Universal Serial Bus Specification 1.1, <http://www.usb.org>
- Universal Serial Bus Specification 2.0, <http://www.usb.org>
- USB device class specifications (Audio, HID, Printer, etc.), <http://www.usb.org>
- USB 2.0, Hrsg. H. Kelm, Franzi's Verlag, 2001, ISBN 3-7723-7965-6



# Chapter 12

## Glossary

---

CPU	Central Processing Unit. The “brain” of a microcontroller; the part of a processor that carries out instructions.
EOT	End Of Transmission.
FIFO	First-In, First-Out.
ISR	Interrupt Service Routine. The routine is called automatically by the processor when an interrupt is acknowledged. ISRs must preserve the entire context of a task (all registers).
RTOS	Real-time Operating System.
Scheduler	The program section of an RTOS that selects the active task, based on which tasks are ready to run, their relative priorities, and the scheduling system being used.
Stack	An area of memory with LIFO storage of parameters, automatic variables, return addresses, and other information that needs to be maintained across function calls. In multitasking systems, each task normally has its own stack.
Superloop	A program that runs in an infinite loop and uses no real-time kernel. ISRs are used for real-time parts of the software.
Task	A program running on a processor. A multitasking system allows multiple tasks to execute independently from one another.
Tick	The OS timer interrupt. Usually equals 1 ms.

# Index

## C

### Core data structures

URB .....	66
USBH_BULK_INT_REQUEST .....	59
USBH_CONTROL_REQUEST .....	58
USBH_ENDPOINT_REQUEST .....	62
USBH_ENUM_ERROR .....	55
USBH_EP_MASK .....	57
USBH_FUNCTION .....	71
USBH_HEADER .....	68
USBH_INTERFACE_INFO .....	54
USBH_INTERFACE_MASK .....	53
USBH_ISO_FRAME .....	60
USBH_ISO_REQUEST .....	61
USBH_PNP_EVENT .....	70
USBH_PNP_NOTIFICATION .....	67
USBH_POWER_STATE .....	74
USBH_SET_CONFIGURATION .....	63
USBH_SET_INTERFACE .....	64
USBH_SET_POWER_STATE .....	65
USBH_SPEED .....	69

### Core function types

USBH_ON_COMPLETION_FUNC .....	78
USBH_ON_ENUM_ERROR_FUNC .....	77
USBH_ON_PNP_EVENT_FUNC .....	76

### Core functions

USBH_CloseInterface() .....	39
USBH_CreateInterfaceList() .....	29
USBH_DestroyInterfaceList() .....	30
USBH_EnumerateDevices() .....	27
USBH_Exit() .....	26
USBH_GetCurrentConfigurationDescriptor() .....	41
USBH_GetDeviceDescriptor() .....	40
USBH_GetEndpointDescriptor() .....	43
USBH_GetFrameNumber() .....	46
USBH_GetInterfaceDescriptor() .....	42
USBH_GetInterfaceID() .....	31
USBH_GetInterfaceIDByHandle() .....	47
USBH_GetInterfaceInfo() .....	32
USBH_GetSerialNumber .....	44
USBH_GetSpeed() .....	45
USBH_GetStatusStr() .....	51

USBH_Init() .....	25
USBH_OpenInterface() .....	38
USBH_RegisterEnumErrorNotification() .....	35
USBH_RegisterPnPNotification() .....	33
USBH_RestartEnumError() .....	37
USBH_SetTraceMask() .....	49
USBH_SubmitUrb() .....	48
USBH_UnregisterEnumErrorNotification() . 36	
USBH_UnregisterPnPNotification() .....	34

## E

### emUSB-Host

Features .....	10
Integrating into your system .....	16

## M

### MSD error codes

USBH_MSD_STATUS_COMMAND_FAILED	96
USBH_MSD_STATUS_ERROR .....	96
USBH_MSD_STATUS_INTERFACE_PROTOCO L .....	96
USBH_MSD_STATUS_INTERFACE_SUB_CLA SS .....	96
USBH_MSD_STATUS_LENGTH .....	96
USBH_MSD_STATUS_PARAMETER .....	96
USBH_MSD_STATUS_PIPE_STALLED ...	96
USBH_MSD_STATUS_SENSE_REPEAT ..	97
USBH_MSD_STATUS_SENSE_STOP .....	97
USBH_MSD_STATUS_SUCCESS .....	96
USBH_MSD_STATUS_TIMEOUT .....	96
USBH_MSD_STATUS_TRANSMISSION ..	96
USBH_MSD_STATUS_WRITE_PROTECT	97

### MSD functions

USBH_MSD_GetLuns() .....	86, 103
USBH_MSD_GetStatus() .....	90
USBH_MSD_GetUnitInfo() .....	89
USBH_MSD_Init() .....	85, 102, 104–105
USBH_MSD_ReadSectors() .....	87
USBH_MSD_UNIT_INFO .....	92, 94, 107
USBH_MSD_WriteSectors() .....	88

**O**

OS integration

API functions ..... 131

**S**

Syntax, conventions used ..... 5