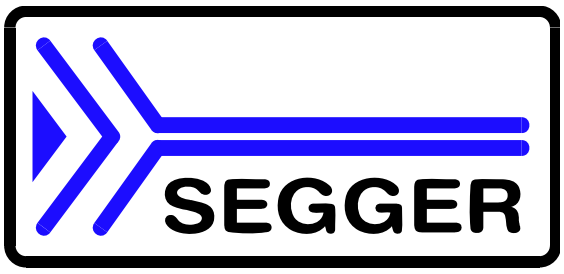
emUSB Host

CPU independent
USB Host stack for
embedded applications

User Guide

Software version 1.00
Document: UM10001
Revision: 0
Date: June 12, 2009



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (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

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
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 Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 1.1: Typographic conventions

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

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

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

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash 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
1.4.1 Related books 12
1.5 Development environment (compiler) 13

2 Running emUSB Host on target hardware 15

2.1 Step 1: Open an embOS 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
3.1.1 emUSB Host HID (TBD FileName) 23

4 USB Host Core Functions 25

4.1 Management Functions 26
4.2 API Functions 30
4.3 Data Structures 54
4.4 Function Types 77
4.5 Use of undocumented functions 81

5 Mass Storage Device 83

5.1 Introduction 84
5.2 Overview 85
5.2.1 Features 85
5.2.2 Restrictions 85
5.3 Supported Protocols 86
5.4 USB Host MSD Core Functions 87
5.4.1 API Functions 87

6 Human Interface Device 101

6.1 TBD 102

7 Configuring emUSB Host 103

7.1 Runtime configuration 104
7.1.1 Driver handling 104
7.2 Compile-time configuration 105
7.2.1 Compile-time configuration switches 105
7.2.2 Debug level 105

8 Debugging 107

8.1 Message output 108
8.2 Testing stability 109
8.3 API functions 110
8.4 Message types 118

9 OS integration 119
9.1 General information 120
9.2 OS layer API functions 121
9.2.1 Examples 121

10 Performance & resource usage 123
10.1 Memory footprint 124
10.1.1 ROM 124
10.1.2 RAM 124
10.2 Performance 125

11 Related Documents 127

12 Glossary 129

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.

1.4.1 Related books

TBD

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 embOS, SEGGER's real-time operating system. We recommend to start with an embOS sample project and include emUSB Host into this project.

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

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

TBD

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 embOS start project

Replace the BSP.c source file and the BSP.h header file used in your embOS 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.

TBD

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.

3.1.1 emUSB Host HID (TBD FileName)

…

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
USBH_Init()	Basically initializes the USB Host stack.
USBH_Exit()	Is called on exit of the library.
USBH_EnumerateDevices()	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 created and added to the bus driver at an 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
USBH_CreateInterfaceList()	Generates a list of available interfaces.
USBH_DestroyInterfaceList()	Deletes a previously generated interface list.
USBH_GetInterfaceID()	Returns the interface ID for a specified interface.
USBH_GetInterfaceInfo()	Obtains information about a specified interface.
USBH_RegisterPnPNotification()	Registers a notification function for PnP events.
USBH_UnregisterPnPNotification()	Unregisters a previously registered notification for PnP events.
USBH_RegisterEnumErrorNotification()	Registers a port error enumeration notification.
USBH_UnregisterEnumErrorNotification()	Unregisters an registered port error enumeration notification.
USBH_RestartEnumError()	The enumeration for all devices that have failed the enumeration is restarted.
USBH_OpenInterface()	Opens the specified interface.
USBH_CloseInterface()	Closes a previously opened interface.
USBH_GetDeviceDescriptor()	Retrieves the device descriptor.
USBH_GetCurrentConfigurationDescriptor()	Retrieves the current configuration descriptor.
USBH_GetInterfaceDescriptor()	Retrieves the interface descriptor.
USBH_GetEndpointDescriptor()	Retrieves a endpoint descriptor.
USBH_GetSerialNumber()	Retrieves the serial number.
USBH_GetSpeed()	Retrieves the operation speed of the device.
USBH_GetFrameNumber()	Retrieves the current frame number.
USBH_GetInterfaceIDByHandle()	Retrieves the current frame number.
USBH_SubmitUrb()	Is used to submit an URB.
USBH_SetTraceMask()	Sets an internal trace mask which filters trace messages produced by the USB bus driver.
USBH_GetStatusStr()	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
InterfaceMask	Input parameter of type USBH_INTERFACE_MASK which specifies a mask for the interfaces which should be listed.
InterfaceCount	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
InterfaceListHandle	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
InterfaceListHandle	Contains the handle for the interface list generated by a call to USBH_CreateInterfaceList.
Index	Specifies the zero based index for an interface in the list.

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
PnPNotification	Contains a pointer to a caller provided structure USBH_PNP_NOTIFICATION. 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
Handle	Contains the valid handle for a PnP notification previously registered by a call to USBH_RegisterPnPNotification.

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
Context	Is a user defined pointer that is passed unchanged to the notification callback function USBH_EnumErrorNotification.
EnumErrorCallback	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 USBH_RegisterEnumErrorNotification.

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
InterfaceID	Specifies the interface to open by its interface ID. The interface ID can be obtained by USBH_PnpNotification or USBH_GetInterfaceID.
Exclusive	Specifies if the interface should be opened exclusive or not. If the value is unequal of zero the interface is opened exclusive.
InterfaceHandle	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 USBH_OpenInterface. 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
Handle	Specifies the interface by its interface handle.
AlternateSetting	Specifies the alternate setting for the interface. The function returns endpoint descriptors that are inside the specified alternate setting.
Mask	Is of type USBH_EP_MASK and specifies a mask to select the endpoint.
Descriptor	Returns a pointer to a caller provided buffer that contains the endpoint descriptor on success.
Size	Specifies the size of the caller provided buffer.
Count	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
Handle	Specifies the interface by its interface handle.
Speed	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
Handle	Specifies the interface by its interface handle.
InterfaceID	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 owner ship of the URB is passed to the bus driver. The storage of the URB must not be freed nor modified as long as the owner ship 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
x	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
USBH_INTERFACE_MASK	Input parameter to create an interface list or to register a PnP notification.
USBH_INTERFACE_INFO	Contains information about a USB interface and the related device.
USBH_ENUM_ERROR	Is used as an notification parameter for the USBH_EnumErrorNotification function.
USBH_EP_MASK	Input parameter to get an endpoint descriptor.
USBH_CONTROL_REQUEST	Is used as a union member for the URB data structure.
USBH_BULK_INT_REQUEST	Is used to transfer data from or to a bulk endpoint.
USBH_ISO_FRAME	Is used to define ISO transfer buffers.
USBH_ISO_REQUEST	Is used to transfer data to an ISO endpoint.
USBH_ENDPOINT_REQUEST	Is used as a union member for the URB data structure.
USBH_SET_CONFIGURATION	Is used as a union member for the URB data structure.
USBH_SET_INTERFACE	Is used as a union member for the URB data structure.
USBH_SET_POWER_STATE	Is used to set a power state.
URB	Basic structure for all asynchronous operations on the bus driver.
USBH_PNP_NOTIFICATION	Is used as an input parameter for the USBH_RegisterPnPNotification function.
USBH_HEADER	Defines the header of an URB.
USBH_SPEED	Is used to get the operation speed of a device.
USBH_PNP_EVENT	Is used as a parameter for the PnP notification.
USBH_FUNCTION	Is used as a member for the USBH_HEADER data structure.
USBH_POWER_STATE	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
Mask	Contains an or'ed selection of the following flags. If the flag is set the related member of this structure is compared to the properties of the USB interface. USBH_INFO_MASK_VID Compare the vendor ID (VID) of the device. USBH_INFO_MASK_PID Compare the product ID (PID) of the device. USBH_INFO_MASK_DEVICE Compare the bcdDevice value of the device. USBH_INFO_MASK_INTERFACE Compare the interface number. USBH_INFO_MASK_CLASS Compare the class of the interface. USBH_INFO_MASK_SUBCLASS Compare the sub class of the interface. USBH_INFO_MASK_PROTOCOL Compare the protocol of the interface.
VID	Contains a vendor ID.
PID	Contains a product ID.
bcdDevice	Contains a BCD coded device version.
Interface	Contains the interface number.
Class	Describes the class code stored in the interface.
Subclass	Describes the sub class code stored in the interface.
Protocol	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
InterfaceID	Contains the unique interface ID. This ID is assigned if the USB device was successful enumerated. It is valid until the device is removed for the host. If the device is reconnected a different interface ID is assigned to each interface.
DeviceID	Contains the unique device ID. This ID is assigned if the USB device was 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.
VID	Contains the vendor ID.
PID	Contains the product ID.
bcdDevice	Contains the BCD coded device version.
Interface	Contains the USB interface number.
Class	Specifies the interface class.
Subclass	Specifies the interface sub class.
Protocol	Specifies the interface protocol.
OpenCount	Specifies the number of open handles for this interface.
ExclusiveUsed	Determines if this interface is used exclusive.
Speed	Specifies the operation speed of this interface.
SerialNumber[256]	Contains the serial number as a counted UNICODE string.
SerialNumberSize	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	Additional flags to determine the location and the type of the error. USBH_ENUM_ERROR_EXTHUBPORT_FLAG means the device is connected to an external hub. USBH_ENUM_ERROR_RETRY_FLAG the bus driver retries the enumeration of this device automatically. USBH_ENUM_ERROR_STOP_ENUM_FLAG the bus driver does not restart the enumeration for this device because all retries has failed. The application can force the bus driver to restart the enumeration by calling the function USBH_RestartEnumError. USBH_ENUM_ERROR_DISCONNECT_FLAG means the device has been disconnected during the enumeration. If the hub port reports a disconnect state the device cannot be re-enumerated by the bus driver automatically. Also the function USBH_RestartEnumError cannot re-enumerate the device. USBH_ENUM_ERROR_ROOT_PORT_RESET means an error during the USB reset of a root hub port occurs. USBH_ENUM_ERROR_HUB_PORT_RESET means an error during a reset of an external hub port occurs. UDB_ENUM_ERROR_INIT_DEVICE means an error during the device initialization (e.g. no answer to a descriptor request or it failed other standard requests. UDB_ENUM_ERROR_INIT_HUB means the enumeration of an external hub fails.

Table 4.24: USBH_ENUM_ERROR() member list

Member	Description
PortNumber	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.
Status	Status of the failed operation.
ExtendedErrorInformation	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: USBH_EP_MASK_INDEX The Index is used for comparison. USBH_EP_MASK_ADDRESS The Address field is used for comparison. USBH_EP_MASK_TYPE The Type field is used for comparison. USBH_EP_MASK_DIRECTION The Direction field is used for comparison.
Index	If valid, this member contains the zero based index of the endpoint in the interface.
Address	If valid, this member contains an endpoint address with direction bit.
Type	If valid, this member specifies a direction. It is one of the following values: USB_IN_DIRECTION USB_OUT_DIRECTION

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

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

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

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

Table 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
`Endpoint`	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
ConfigurationDescriptorIndex	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
`AlternateSetting`	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
PowerState	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
ControlRequest	USBH_CONTROL_REQUEST
BulkIntRequest	USBH_BULK_INT_REQUEST
IsoRequest	USBH_ISO_REQUEST
EndpointRequest	USBH_ENDPOINT_REQUEST
SetConfiguration	USBH_SET_CONFIGURATION
SetInterface	USBH_SET_INTERFACE
SetPowerState	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
Header	Contains the URB header of type USBH_HEADER. The most important parameters are the function code and the callback function.
Request	Is a union and contains information depending on the specific request of the USBH_HEADER.

Table 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
PnpNotification	Contains the notification function that is called from the library if a PnP event occurs.
Context	Contains the notification context that is passed unchanged to the notification function.
PowerState	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 USBH_SubmitUrb returns USBH_STATUS_PENDING. 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
USBH_SPEED_UNKNOWN	The speed is unknown.
USBH_LOW_SPEED	The device operates at low speed.
USBH_FULL_SPEED	The device operates at full speed.
USBH_HIGH_SPEED	The device operates at high speed.

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

Table 4.39: USBH_FUNCTION() member list

Entry	Description
USBH_FUNCTION_RESET_DEVICE	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.
USBH_FUNCTION_RESET_ENDPOINT	Clears an error condition on a special endpoint. If a data transfer error occurs that cannot be handled in hardware the bus driver stops the endpoint and does not allow further data transfers before the endpoint is 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 USBH_FUNCTION_ABORT_ENDPOINT. This function uses the data structure USBH_ENDPOINT_REQUEST.
USBH_FUNCTION_ABORT_ENDPOINT	Aborts all pending requests on a endpoint. The host controller calls the completion function with a status code USBH_STATUS_CANCELED. The completion of the URBs may be delayed. The application should wait until all pending requests has been returned by the driver before the handle is closed or USBH_FUNCTION_RESET_ENDPOINT is called.

Table 4.39: USBH_FUNCTION() member list

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

Table 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
USBH_NORMAL_POWER	The device is switched to normal operation.
USBH_SUSPEND	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
Context	Is the user defined pointer that was passed to USBH_RegisterPnPNotification. The library does not modify this parameter.
Event	Specifies the PnP event.
InterfaceID	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
Urb	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

5.1 Introduction

The USB Host MSD library is a generic firmware library for accessing USB Mass Storage Devices. It implements the USB Mass Storage Device class protocols specified by the USB Implementers Forum. It maps read/write requests issued by a file system driver to protocol-specific SCSI-style commands. It also implements initialization, discovery and error recovery. The library is designed to be easy to use and provides a convenient programming interface.

This document describes the architecture, the features and the programming interface of the code. Furthermore, it includes instructions for including the library in a firmware project.

Throughout this document the software layer that directly attaches to USB Host library is called "main program", regardless of whether it is the main loop of a simple firmware or a task of an operating system.

The reader of this document is assumed to be familiar with the specification of the Universal Serial Bus Version 1.1 and 2.0 as well as common aspects of C programming.

5.2 Overview

The USB Host library handles all necessary commands and protocols to access a USB Mass Storage device. It provides an easy to use interface for integrating the library in an application. For more information refer also to section 3.

5.2.1 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 from the file system. Support for a file system depends on the used file system library.

5.2.2 Restrictions

The following restrictions relate to the USB Host library:

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

5.3 Supported Protocols

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

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

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

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

5.4 USB Host MSD Core Functions

5.4.1 API Functions

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

Function	Description
USBH_MSD_Init()	Initializes the USBH MSD library.
USBH_MSD_AddDevice()	Adds a USB device to the library.
USBH_MSD_RemoveDevice()	Removes a USB device from the library.
USBH_MSD_GetLuns()	Returns an array of logical unit numbers (LUN's).
USBH_MSD_ReadSectors()	Reads sectors from a USB Mass Storage device.
USBH_MSD_WriteSectors()	Writes sectors to a USB Mass Storage device.
USBH_MSD_GetUnitInfo()	Returns basic information about the logical unit (LUN).
USBH_MSD_GetStatus()	Checks the state of a device unit.

Table 5.1: emUSB Host MSD API function overview

5.4.1.1 USBH_MSD_Init()

Description

Initializes the USBH MSD library.

Prototype

```
void USBH_MSD_Init();
```

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.

5.4.1.2 USBH_MSD_AddDevice()

Description

Adds a USB device to the library.

Prototype

```
int USBH_MSD_AddDevice();
```

Return Value

If successful the function returns a value >= 0 describing the zero based device index. It returns a value < 0 to indicate an error. Please note that the device index is different from the logical unit index which is returned by the function USBH_MSD_GetLuns. The device index is only required for the functions USBH_MSD_GetLuns and USBH_MSD_RemoveDevice.

Additional information

The function USBH_MSD_Init has to be called before with success. USBH_MSD_AddDevice has to be called before a device operation is performed. The library checks the device for a valid transport method and protocol. If the library does not support the protocol or the transport method of this device an appropriate error code is returned. The library accepts only interfaces that have a particular sub class code and protocol code. This function can be used to test unknown devices. Use USBH_MSD_RemoveDevice to remove an unused device.

5.4.1.3 USBH_MSD_RemoveDevice()

Description

Removes a USB device from the library.

Prototype

```
int USBH_MSD_RemoveDevice( int DevIndex );
```

Parameter

Parameter	Description
DevIndex	The device index returned by the function USBH_MSD_AddDevice.

Table 5.2: USBH_MSD_RemoveDevice() parameter list

Return Value

If successful the function returns USBH_MSD_STATUS_SUCCESS. If parameter DevIndex points to an invalid device USBH_MSD_STATUS_ERROR is returned.

Additional information

USBH_MSD_RemoveDevice should only be called if no operation is pending because the library will not send special requests to the device to reset it.

5.4.1.4 USBH_MSD_GetLuns()

Description

Returns an array of logical unit numbers (LUN's).

Prototype

```
int USBH_MSD_GetLuns(
    int          DevIndex,
    U8           LunArray[],
    unsigned int LunArraySize );
```

Parameters

Parameter	Description
DevIndex	The device index returned by the function USBH_MSD_AddDevice.
LunArray[]	A pointer to a caller provided storage. The storage is handled as an array of U8, where each value represents the index of a LUN. These LUN indexes are required for accessing the file system.
LunArraySize	Contains the size of the LunArray.

Table 5.3: USBH_MSD_GetLuns() parameter list

Return Value

If successful the function returns a value >= 0 describing the number of valid LUN's in the array. It returns a value < 0 to indicate an error.

Additional information

USBH_MSD_AddDevice must be called before with success.
The function returns an array of valid LUN indexes. For each LUN an instance of the file system can be started. The LUN index is a required parameter for the functions USBH_MSD_ReadSectors, USBH_MSD_WriteSectors, USBH_MSD_GetUnitInfo and USBH_MSD_GetStatus.

5.4.1.5 USBH_MSD_ReadSectors()

Description

Reads sectors from a USB Mass Storage device.

Prototype

```
void USBH_MSD_ReadSectors(
     U8    Lun,
     U32   SectorAddress,
     U32   NumSectors,
     U8  * Buffer );
```

Parameters

Parameter	Description
Lun	Logical unit number returned by a call to USBH_MSD_GetLuns.
SectorAddress	Describes the first sector to read.
NumSectors	Determines the number of sectors to read.
Buffer	Pointer to a byte buffer. The caller is responsible for the storage of the buffer.

Table 5.4: 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.

Additional information

A valid LUN has to be requested by a call to USBH_MSD_GetLuns before you are able to successfully call USBH_MSD_ReadSectors.

5.4.1.6 USBH_MSD_WriteSectors()

Description

Writes sectors to a USB Mass Storage device.

Prototype

```
void USBH_MSD_WriteSectors(
     U8    Lun,
     U32   SectorAddress,
     U32   NumSectors,
     U8  * Buffer );
```

Parameters

Parameter	Description
Lun	Logical unit number returned from USBH_MSD_GetLuns.
SectorAddress	Describes the first sector to write.
NumSectors	Determines the number of sectors to write.
Buffer	Pointer to a buffer containing the data to be written.

Table 5.5: 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.

Additional information

Can be called after a valid LUN was requested by a call to USBH_MSD_GetLuns.

5.4.1.7 USBH_MSD_GetUnitInfo()

Description

Returns basic information about the logical unit (LUN).

Prototype

```
int USBH_MSD_GetUnitInfo(
    U8                    Lun,
    USBH_MSD_UNIT_INFO * Info );
```

Parameters

Parameter	Description
Lun	Logical unit number returned from USBH_MSD_GetLuns.
Info	Pointer to a caller provided storage buffer. It will contain the information about the LUN in case of success.

Table 5.6: 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.

Additional information

Can be called after a valid LUN was requested by a call to USBH_MSD_GetLuns.

5.4.1.8 USBH_MSD_GetStatus()

Description

Checks the state of a device unit.

Prototype

int USBH_MSD_GetStatus(U8 Lun);

Parameter

Parameter	Description
Lun	Logical unit number returned from USBH_MSD_GetLuns.

Table 5.7: 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.

Additional information

Can be called after a valid LUN was requested by a call to USBH_MSD_GetLuns.

5.4.2 Data Structures

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

Structure	Description
USBH_MSD_UNIT_INFO	Contains logical unit information.

Table 5.8: emUSB Host MSD structure overview

5.4.2.1 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 5.9: USBH_MSD_UNIT_INFO() parameter list

5.4.3 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 5.10: emUSB Host MSD error code overview

5.4.3.1 USBH_MSD_STATUS_SUCCESS

Description

The operation has been successfully completed.

5.4.3.2 USBH_MSD_STATUS_ERROR

Description

The operation has been completed with an error.

5.4.3.3 USBH_MSD_STATUS_PARAMETER

Description

A parameter is incorrect.

5.4.3.4 USBH_MSD_STATUS_LENGTH

Description

The operation detected a length error.

5.4.3.5 USBH_MSD_STATUS_TIMEOUT

Description

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

5.4.3.6 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.

5.4.3.7 USBH_MSD_STATUS_INTERFACE_PROTOCOL

Description

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

5.4.3.8 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.

5.4.3.9 USBH_MSD_STATUS_PIPE_STALLED

Description

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

5.4.3.10 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.

5.4.3.11 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.

5.4.3.12 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.

5.4.3.13 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

6.1 TBD

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	USBH_DEBUG	0	Macro to define the debug level of the emUSB Host build.
Optimization macros			
F	USBH_MEMCPY	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	USBH_MEMSET	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	USBH_MEMMOVE	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	USBH_MEMCMP	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 Testing stability

TBD

8.3 API functions

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

Table 8.1: emUSB Host debugging API function overview

8.3.1 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
FilterMask	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.

8.3.2 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
FilterMask	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.

8.3.3 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
FilterMask	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.

8.3.4 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
FilterMask	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.

8.3.5 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
s	TBD

Table 8.6: USBH_LOG() parameter list

8.3.6 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
s	TBD

Table 8.7: USBH_WARN() parameter list

8.3.7 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 );
```

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.

8.4 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 8.8: 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 IP/OS interface. For embOS, all functions required for this IP/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 embOS are implemented in `USBH_OS_embOS.c` which is located in the root folder of the IP stack.

9.2 OS layer API functions

Function	Description
General macros	
USBH_OS_Delay()	Blocks the calling task for a given time.
USBH_OS_DisableInterrupt()	Disables interrupts.
USBH_OS_EnableInterrupt()	Enables interrupts.
USBH_OS_GetTime32()	Returns the current system time in ticks. 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.
USBH_OS_Init()	Creates and initializes all objects required for task synchronization. These are 2 events (for USBH_Task and USBH_RxTask) and one semaphore for protection of critical code which may not be executed from multi-ple task at the same time.
USBH_OS_Lock()	The stack requires a single lock, typically a resource semaphore or mutex. This function locks this object, guarding sections of the stack code against other tasks. If the entire stack executes from a single task, no functionality is required here.
USBH_OS_Unlock()	Unlocks the single lock used locked by a previous call to USBH_OS_Lock().
USBH_Task synchronization	
USBH_OS_SignalNetEvent()	Wakes the USBH_Task if it is waiting for a NET-event or timeout in the function USBH_OS_WaitNetEvent().
USBH_OS_WaitNetEvent()	Called from USBH_Task only. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalNetEvent() is called from an other task or ISR.
USBH_RxTask synchronization	
USBH_OS_SignalRxEvent()	Wakes the USBH_RxTask if it is waiting for a NET-event or timeout in the function USBH_OS_WaitRxEvent().
USBH_OS_WaitRxEvent()	Optional. Called from USBH_RxTask, if it is used to receive data. Blocks until the timeout expires or a NET-event occurs, meaning USBH_OS_SignalRxEvent() is called from the ISR.
Application task synchronization	
USBH_OS_WaitItem()	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.
USBH_OS_WaitItemTimed()	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.
USBH_OS_SignalItem()	Sets an event object to signaled state, or resumes tasks which are waiting at the event object. Function is called from a task, not an ISR.

Table 9.1: Target OS interface function list

9.2.1 Examples

OS interface routine for embOS

All OS interface routines are implemented in `USBH_OS_embOS.c` which is located in the root folder of the IP 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
Bulk	
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 68
- USBH_BULK_INT_REQUEST 61
- USBH_CONTROL_REQUEST 60
- USBH_ENDPOINT_REQUEST 64
- USBH_ENUM_ERROR 57
- USBH_EP_MASK 59
- USBH_FUNCTION 73
- USBH_HEADER 70
- USBH_INTERFACE_INFO 56
- USBH_INTERFACE_MASK 55
- USBH_ISO_FRAME 62
- USBH_ISO_REQUEST 63
- USBH_PNP_EVENT 72
- USBH_PNP_NOTIFICATION 69
- USBH_POWER_STATE 76
- USBH_SET_CONFIGURATION 65
- USBH_SET_INTERFACE 66
- USBH_SET_POWER_STATE 67
- USBH_SPEED 71

Core function types
- USBH_ON_COMPLETION_FUNC 80
- USBH_ON_ENUM_ERROR_FUNC 79
- USBH_ON_PNP_EVENT_FUNC 78

Core functions
- USBH_CloseInterface() 41
- USBH_CreateInterfaceList() 31
- USBH_DestroyInterfaceList() 32
- USBH_EnumerateDevices() 29
- USBH_Exit() 28
- USBH_GetCurrentConfigurationDescriptor() 43
- USBH_GetDeviceDescriptor() 42
- USBH_GetEndpointDescriptor() 45
- USBH_GetFrameNumber() 48
- USBH_GetInterfaceDescriptor() 44
- USBH_GetInterfaceID() 33
- USBH_GetInterfaceIDByHandle() 49
- USBH_GetInterfaceInfo() 34
- USBH_GetSerialNumber 46
- USBH_GetSpeed() 47
- USBH_GetStatusStr() 53
- USBH_Init() 27
- USBH_OpenInterface() 40
- USBH_RegisterEnumErrorNotification() 37
- USBH_RegisterPnPNotification() 35
- USBH_RestartEnumError() 39
- USBH_SetTraceMask() 51
- USBH_SubmitUrb() 50
- USBH_UnregisterEnumErrorNotification() ..38
- USBH_UnregisterPnPNotification() 36

E

emUSB Host
- Features 10
- Integrating into your system 16

M

MSD error codes
- USBH_MSD_STATUS_COMMAND_FAILED 99
- USBH_MSD_STATUS_ERROR 99
- USBH_MSD_STATUS_INTERFACE_PROTOCOL 99
- USBH_MSD_STATUS_INTERFACE_SUB_CLASS 99
- USBH_MSD_STATUS_LENGTH 99
- USBH_MSD_STATUS_PARAMETER 99
- USBH_MSD_STATUS_PIPE_STALLED ... 99
- USBH_MSD_STATUS_SENSE_REPEAT .100
- USBH_MSD_STATUS_SENSE_STOP 100
- USBH_MSD_STATUS_SUCCESS 99
- USBH_MSD_STATUS_TIMEOUT 99
- USBH_MSD_STATUS_TRANSMISSION .. 99
- USBH_MSD_STATUS_WRITE_PROTECT 100

MSD functions
- USBH_MSD_AddDevice() 89
- USBH_MSD_GetLuns() 91
- USBH_MSD_GetStatus() 95
- USBH_MSD_GetUnitInfo() 94
- USBH_MSD_Init() 88
- USBH_MSD_ReadSectors() 92
- USBH_MSD_RemoveDevice() 90
- USBH_MSD_UNIT_INFO 97

USBH_MSD_WriteSectors() 93

O

OS integration
API functions 121

S

Syntax, conventions used 5

