

emUSB

CPU independent
USB Device stack for
embedded applications

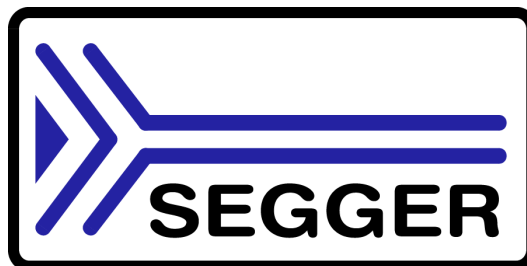
User Guide

Software version 2.34

Document: UM09001

Revision: 1

Date: November 16, 2011



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

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

© 2007 SEGGER Microcontroller GmbH & Co. KG, Hilden / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available.

For registration, please provide the following:

- Company name and address
- Your name
- Your job title
- Your email address and telephone number
- Name and version of the product

Please send this information to: register@segger.com

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 software version 2.00. 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.

Manual Change History

Manual version	Date	By	Explanation
2.34/01	111116	YR	Updated USB Core chapter: * Added description for function: USB__WriteEP0FromISR() Removed some typos.
2.34/00	111111	SR	Updated CDC chapter: * Added new function: USB_CDC_SetOnBreak() * Updated the functions: USB_MSD_INST_DATA_DRIVER Chapter Target USB driver: * Added new drivers to the list. Removed some typos.
2.32/00	101206	SR	Added new functions in USB Core chapter: * USB_SetVendorRequestHook(), USB_SetIsSelfPowered() Updated the Product Ids in Chapter GettingTheTargetUp\Confi- guration. Updated MSD chapter: * Added new picture on front page. * Updated chapter Overview * Added new function: USB_MSD_Connect(), USB_MSD_Disconnect(), USB_MSD_RequestDisconnect(), USB_MSD_UpdateWriteProtect(), USB_MSD_WaitForDisconnection(), * Updated the functions: USB_MSD_INST_DATA_DRIVER Updated the CDC chapter: * Added new Ex-Functions * Added new serial status functions. Added new picture to the front page of chapter HID. Update Printer Class chapter: * Added new picture to the front page to the chapter * Added new information to the USB_PRINTER_API. Chapter Target USB driver: * Added new drivers to the list.
2.30/00	101022	SR	Added the function for remote wakeup.
2.27/00	100730	MD	Chapter "Printer Class" added.
2.26/01	090127	SR	Chapter USB core: * Added new functions: USB_SetMaxPower(), USB_SetOnRxEP0(), USB_SetOnSetupHook() Chapter Bulk Communication: * Added new functions: * USB_BULK_CancelRead() * USB_BULK_CancelWrite() * USB_BULK_ReadTimed() * USB_BULK_SetOnRXHook() * USB_BULK_WaitForTX() * USB_BULK_WaitForRX() * USB_BULK_WriteEx() * USB_BULK_WriteExTimed * USB_BULK_WriteNULLPacket() * USB_BULK_WriteTimed(). Chapter CDC: * Added new functions: * USB_CDC_CancelRead() * USB_CDC_CancelWrite() * USB_CDC_ReadTimed() * USB_CDC_ReceiveTimed() Updated indexes in chapter CDC, Bulk communication, MSD, HID.
2.22/01	080917	SR/ SK	Added new chapter Combining different USB components (Multi-Interface) All chapter reviewed and cleaned up.

Manual version	Date	By	Explanation
2.22/00	080902	SR	Chapter USB core: * Added new function USB_EnableIAD. Chapter Bulk communication: * Update description of USB_BULK_Receive. Chapter MSD component: * Updated "Final configuration". * Updated "Class specific configuration functions." Chapter CDC component: * Added new functions: USB_CDC_ReadOverlapped(), USB_CDC_WriteOverlapped(), USB_CDC_WaitForRx, USB_CDC_WaitForTx(). Chapter Target USB driver: * Updated available driver list. Chapter FAQ: * Added new
15.0	080403	SR	Update company's address and legal form.
14.0	071204	SR	Chapter "Target USB driver": * Updated "Writing your own driver": - pfStallEP changed to pfSetClrStallEP. - Added new driver ST STR91x. - Added descripton for pfResetEP. Chapter "Bulk Communication": * Added new function: USB_BULK_Receive() * Added new function: USB_BULK_GetNumBytesInBuffer()
13.0	071005	SK	Chapter "Target USB driver": * Section "Interrupt handling" added.
12.0	070706	SR	Chapter "USB core": * Changed USB_GetStastus to USB_GetState Chapter "MSD": * "MSD_Start.c" changed to "MSD_Start_StorageRAM.c" * Added information to "USB_MSD_INST_DATA_DRIVER" * "Storage drivers supplied with this release" updated. Chapter "Bulk communication": * Changed text for USBBULK_GetMode/Ex.
11.0	070704	SK	Chapter "Introduction": * HID section added. Chapter "USB Core": * USB_GetState() added. Chapter "HID": * USBHID_Init() updated. Chapter "Target OS Interface": * USB_OS_RestoreI() removed. * USB_OS_DI() removed. Chapter "Target USB Driver": * STR750 added.
10.0	070618	SK	Chapter "HID" added. Chapter "USB Core" added. Chapter "Bulk communication": * USB core functions removed. Chapter "Introduction": * Section "Development environment" added.
9.0	070123	SK	emUSB components renamed: * "emUSB with bulk component" to "emUSB-Bulk" * "emUSB with MSD component" to "emUSB-MSD" * "emUSB with CDC component" to "emUSB-CDC" Chapter "Introduction": * updated and enhanced * emUSB-CDC added
8.0	070121	SK	Product name changed from "USB-Stack" to "emUSB". Various changes in layout and structure. Chapter "About" added. Chapter "Introduction": * updated * "emUSB structure" graphic added. Chapter "Bulk communication": * USB_SetClassRequestHook(): - Function description added. Chapter "CDC": * Head of description of USB_CDC_LINE_CODING changed.

Manual version	Date	By	Explanation
7.0	070109	SR	Added new chapter CDC.
6.0	061221	SR	Added new USBBulk HOST-API function USBBULK_SetUSBId(). Company description added
5.0	061220	SR	Changed chapter 1.1.1 USB-Bulk stack: Info reg. availability of the Host-driver source. Updated chapter title "Getting the target up" Updated chapter 1.1.2.3 Features Updated chapter 1 - Information of max. data transfer rates updated.
4.0	061212	SR	Added chapter "Mass Storage Device" Changed chapter Background info: -Updated Changed chapter title "Configuring the target" to "Getting the target up" Moved any related information of files provided with the USB stack to "Getting the target up"
3.0	061120	SR	Added the extended HOST API functionality to manual
2.0	061115	SR	Updated chapter: Target USB driver Bulk Communication
1.0	060808	OO	Initial Version

About this document

Required knowledge

C Programming:

This guide assumes that you already possess a solid knowledge of the C programming language. If you feel that your knowledge of C is not sufficient, we recommend "*The C Programming Language*" by Kernighan and Ritchie, which describes the programming standard, and—in newer editions—also covers the ANSI C standard. Knowledge of assembly programming is not required.

USB:

There is usually no need to have an intimate knowledge of USB internals. However, a basic understanding of USB is expected. The chapter *Background information* on page 29 provides information about USB in general and the mass storage class in particular and might be of interest if you are new to USB.

How to use this manual

This manual explains all the functions and macros that emUSB 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.
Reference	Reference to chapters, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 1.1: Typographic conventions

Data types

Because C does not provide data types of fixed lengths which are identical on all platforms, the stack uses, in most cases, its own data types as shown in the table below:

Data type	Definition	Explanation
I8	signed char	8-bit signed value
U8	unsigned char	16-bit unsigned value
I16	signed short	16-bit signed value
U16	unsigned short	16-bit unsigned value
I32	signed long	32-bit signed value
U32	unsigned long	32-bit unsigned value

Table 1.1: Data type

For most 16/32-bit controllers, the C data types used will work fine. However, the data types can be configured in the main configuration file.



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.



emUSB
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	15
1.1	Overview	16
1.2	emUSB features	16
1.3	emUSB components	17
1.3.1	emUSB-Bulk	18
1.3.1.1	Purpose of emUSB-Bulk	18
1.3.2	emUSB-MSD	19
1.3.2.1	Purpose of emUSB-MSD	19
1.3.2.2	Typical applications	19
1.3.2.3	emUSB-MSD features	19
1.3.2.4	How does it work?	19
1.3.3	emUSB-CDC	21
1.3.3.1	Typical applications	21
1.3.4	emUSB-HID	22
1.3.4.1	Typical applications	22
1.3.5	emUSB-Printer	23
1.3.5.1	Typical applications	23
1.4	Requirements	24
1.4.1	Target system	24
1.4.2	Development environment (compiler)	24
1.5	File structure	25
1.5.1	Bulk communication component	26
1.5.2	MSD component	26
1.5.3	CDC component	26
1.5.4	HID component	26
2	Background information	29
2.1	USB	30
2.1.1	Short Overview	30
2.1.2	Important USB Standard Versions	30
2.1.3	USB System Architecture	31
2.1.4	Transfer Types	33
2.1.5	Setup phase / Enumeration	33
2.1.6	Product / vendor IDs	33
2.2	Predefined device classes	34
2.3	USB analyzers	34
2.4	References	34
3	Getting started	35
3.1	How to setup your target system	36
3.1.1	Upgrade a trial version available on the web with source code.	36
3.1.2	Upgrading an embOS Start project	37
3.1.3	Creating a project from scratch	38
3.2	Select the start application	40
3.3	Build the project and test it	40
3.4	Configuration	41
3.4.1	General emUSB configuration functions	42
3.4.2	Additional required configuration functions for emUSB-MSD	47
3.4.3	Descriptors	47

4	USB Core.....	49
4.1	Overview	50
4.2	Target API.....	51
4.2.1	USB basic functions	52
4.2.2	USB configuration functions	57
4.2.3	USB control functions	66
4.2.4	USB IAD functions.....	68
4.2.5	USB Remote wakeup functions.....	69
5	Bulk communication.....	73
5.1	Generic bulk stack.....	74
5.2	The Kernel mode driver (PC).....	74
5.2.1	Why is a driver necessary?	74
5.2.2	Supported platforms.....	74
5.3	Installing the driver	74
5.3.1	Recompiling the driver	77
5.3.2	The .inf file.....	78
5.3.3	Configuration.....	79
5.4	Example application.....	80
5.4.1	Running the example applications.....	81
5.4.2	Compiling the PC example application	83
5.5	Target API.....	84
5.5.1	Target interface function list	85
5.5.2	USB-Bulk functions.....	86
5.5.3	Data structures.....	106
5.6	Host API	108
5.6.1	Host API list	109
5.6.2	USB-Bulk Basic functions.....	111
5.6.3	USB-Bulk direct input/output functions.....	115
5.6.4	USB-Bulk Control functions.....	121
6	Mass Storage Device Class (MSD)	139
6.1	Overview	140
6.2	Configuration.....	141
6.2.1	Initial configuration	141
6.2.2	Final configuration.....	141
6.2.3	Class specific configuration functions	141
6.2.4	Running the example application	146
6.2.4.1	MSD_Start_StorageRAM.c in detail	146
6.3	Target API.....	147
6.3.1	API functions	148
6.3.2	Extended API functions	154
6.3.3	Data structures.....	159
6.4	Storage Driver	166
6.4.1	General information.....	166
6.4.1.1	Supported storage types	166
6.4.1.2	Storage drivers supplied with this release	166
6.4.2	Interface function list.....	166
6.4.3	USB_MSD_STORAGE_API in detail.....	167
7	Communication Device Class (CDC).....	175
7.1	Overview	176
7.1.1	Configuration.....	176
7.2	The example application.....	177
7.3	Installing the driver	180
7.3.1	The .inf file.....	183
7.3.2	Installation verification.....	184
7.3.3	Testing communication to the USB device.....	185
7.4	Target API.....	188

7.4.1	Interface function list	188
7.4.2	API functions	189
7.4.3	Data structures	206
8	Human Interface Device Class (HID)	211
8.1	Overview	212
8.1.1	Further reading	212
8.1.2	Categories	213
8.1.2.1	“True HIDs”	213
8.1.2.2	“Vendor specific HIDs”	213
8.2	Background information	214
8.2.1	HID descriptors	214
8.2.1.1	HID descriptor.....	214
8.2.1.2	Report descriptor.....	214
8.2.1.3	Physical descriptor.....	215
8.3	Configuration	216
8.3.1	Initial configuration	216
8.3.2	Final configuration	216
8.4	Example application	217
8.4.1	HID_Mouse.c	217
8.4.1.1	Running the example	217
8.4.2	HID_Echo1.c.....	218
8.4.2.1	Running the example	218
8.4.2.2	Compiling the PC example application	219
8.5	Target API	220
8.5.1	Target interface function list.....	220
8.5.2	USB-HID functions.....	221
8.5.3	Data structures	224
8.6	Host API.....	225
8.6.1	Host API function list.....	226
8.6.2	USB-HID functions.....	227
9	Printer Class	239
9.1	Overview	240
9.1.1	Configuration	240
9.2	The example application	241
9.3	Target API	244
9.3.1	Interface function list	244
9.3.2	API functions	245
9.3.3	Data structures	247
10	Combining different USB components (Multi-Interface).....	249
10.1	Overview	250
10.1.1	Single interface device classes.....	251
10.1.2	Multiple interface device classes	252
10.1.3	IAD class.....	252
10.2	Configuration	253
10.3	How to combine	254
10.4	emUSB component specific modification	256
10.4.1	BULK communication component	256
10.4.1.1	Device side	256
10.4.1.2	Host side	256
10.4.2	MSD component	258
10.4.2.1	Device side	258
10.4.2.2	Host side	258
10.4.3	CDC component	258
10.4.3.1	Device side	258
10.4.3.2	Host side	258
10.4.4	HID component	260
10.4.4.1	Device side	260

10.4.4.2	Host side	260
11	Target OS Interface	261
11.1	General information.....	262
11.1.1	Operating system support supplied with this release	262
11.2	Interface function list.....	263
11.3	Example	273
12	Target USB Driver.....	277
12.1	General information.....	278
12.1.1	Available USB drivers.....	278
12.2	Adding a driver to emUSB	280
12.3	Interrupt handling	283
12.3.1	ARM7 / ARM9 based cores	283
12.3.1.1	ARM specific IRQ handler	284
12.3.1.2	Device specifics ATMEL AT91CAP9x.....	285
12.3.1.3	Device specifics ATMEL AT91RM9200	285
12.3.1.4	Device specifics ATMEL AT91SAM7A3	285
12.3.1.5	Device specifics ATMEL AT91SAM7S64, AT91SAM7S128, AT91SAM7S256	285
12.3.1.6	Device specifics ATMEL AT91SAM7X64, AT91SAM7X128, AT91SAM7X256	285
12.3.1.7	Device specifics ATMEL AT91SAM7SE	285
12.3.1.8	Device specifics ATMEL AT91SAM9260	285
12.3.1.9	Device specifics ATMEL AT91SAM9261	285
12.3.1.10	Device specifics ATMEL AT91SAM9263	286
12.3.1.11	Device specifics ATMEL AT91SAMRL64, AT91SAMR64	286
12.3.1.12	Device specifics NXP LPC214x	287
12.3.1.13	Device specifics NXP LPC23xx	287
12.3.1.14	Device specifics NXP (formerly Sharp) LH79524/5	287
12.3.1.15	Device specifics OKI 69Q62	287
12.3.1.16	Device specifics ST STR71x	287
12.3.1.17	Device specifics ST STR750	287
12.3.1.18	Device specifics ST STR750	287
12.4	Writing your own driver	288
12.4.1	USB initialization functions	289
12.4.2	General USB functions	290
12.4.3	General endpoint functions	292
12.4.4	Endpoint 0 (control endpoint) related functions	295
12.4.5	OUT-endpoint functions.....	296
12.4.6	IN-endpoint functions	297
12.4.7	USB driver interrupt handling.....	299
13	Support	301
13.1	Problems with tool chain (compiler, linker)	302
13.1.1	Compiler crash.....	302
13.1.2	Compiler warnings.....	302
13.1.3	Compiler errors.....	302
13.1.4	Linker problems	302
13.2	Problems with hardware/driver	303
13.3	Contacting support	303
14	Performance & resource usage	305
14.1	Memory footprint	306
14.1.1	ROM	306
14.1.2	RAM	306
14.2	Performance	307
15	FAQ.....	309

Chapter 1

Introduction

This chapter will give a short introduction to emUSB, covering generic bulk, Mass Storage Device (MSD), Communication Device Class (CDC), Human Interface Device (HID) and Printer Class functionality. Host and target requirements are covered as well.

1.1 Overview

This guide describes how to install, configure and use emUSB with bulk communication, MSD, CDC or HID component. It also explains the internal structure of emUSB.

emUSB has been designed to work on any embedded system with USB client controller. It can be used with USB 1.1. or USB 2.0 devices.

The highest possible transfer rate on USB 2.0 full speed (12 Mbit/second) devices is approximately 1 Mbyte per second. This data rate can indeed be achieved on fast systems, such as ARM7 and faster.

USB 2.0 high speed mode (480 MBit/second) is also fully supported and is automatically handled. Using USB high speed mode with an ARM9 or faster could achieve values of approx. 18 MBytes/second and faster.

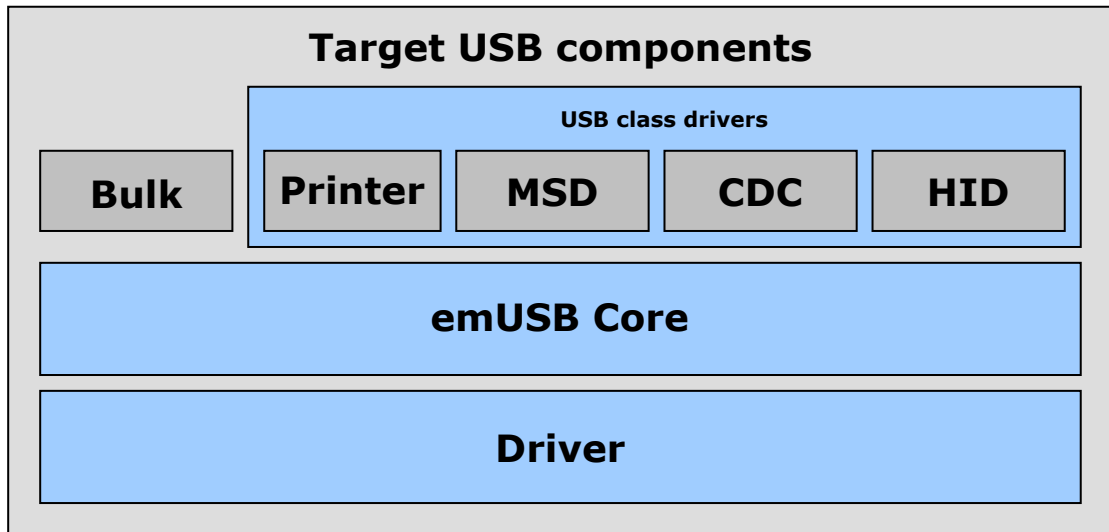
1.2 emUSB features

Key features of emUSB are:

- High speed
- Can be used with or without an RTOS.
- Easy to use
- Easy to port
- No custom USB host driver necessary
- Start / test application supplied
- Highly efficient, portable, and commented ANSI C source code
- Hardware abstraction layer allows rapid addition of support for new devices.

1.3 emUSB components

emUSB consists of three layers: A driver for hardware access, the emUSB core and at least a USB class driver or the bulk communication component.



The different available hardware drivers, the USB class drivers, and the bulk communication component are additional packages, which can be combined and ordered as they fit to the requirements of your project. Normally, emUSB consists of a driver that fits to the used hardware, the emUSB core and at least one of the USB class drivers MSD, CDC, HID, printer or the unclassified bulk component.

Component	Description
USB protocol layer	
Bulk	emUSB bulk component. (emUSB-Bulk)
MSD	emUSB Mass Storage Device class component. (emUSB-MSD).
CDC	emUSB Communication Device Class component. (emUSB-CDC)
HID	emUSB Human Interface Device Class component. (emUSB-HID)
Printer	emUSB Printer Class component. (emUSB-Printer)
Core layer	
emUSB-Core	The emUSB core is the intrinsic USB stack.
Hardware layer	
Driver	USB controller driver.

Table 1.1: emUSB components

1.3.1 emUSB-Bulk

The emUSB-Bulk stack consists of an embedded side, which is shipped as source code, and a driver for the PC, which is typically shipped as an executable (`.sys`). (The source of the PC driver can also be ordered.)

1.3.1.1 Purpose of emUSB-Bulk

emUSB-Bulk allows you to quickly and smoothly develop software for an embedded device that communicates with a PC via USB. The communication is like a single, high speed, reliable channel (very similar to a TCP connection). It basically allows the PC to send data to the embedded target, the embedded target to receive these bytes and reply with any number of bytes. The PC is the USB host, the target is the USB client. The USB standard defines 4 types of communication: Control, isochronous, interrupt, and bulk. Experience shows, that for most embedded devices the bulk mode is the communication mode of choice. It allows usage of the full bandwidth of the USB bus.

1.3.2 emUSB-MSD

1.3.2.1 Purpose of emUSB-MSD

Access the target device like an ordinary disk drive

emUSB-MSD enables the use of an embedded target device as a USB mass storage device. The target device can be simply plugged-in and used like an ordinary disk drive, without the need to develop a driver for the host operating system. This is possible because the mass storage class is one of the standard device classes, defined by the USB Implementers Forum (USB IF). Virtually, every major operating system on the market supports these device classes out of the box.

No custom host drivers necessary

Every major OS already provides host drivers for USB mass storage devices, there is no need to implement your own. The target device will be recognized as a mass storage device and can be accessed directly.

Plug and Play

Assuming the target system is a digital camera using emUSB-MSD, videos or photos taken by this camera can be conveniently accessed with the file system explorer of the used operating system, if the camera is connect to the host.

1.3.2.2 Typical applications

Typical applications are:

- Digital camera
- USB stick
- MP3 player
- DVD player

Any target with USB interface: easy access to configuration and data files.

1.3.2.3 emUSB-MSD features

Key features of emUSB-MSD are:

- Can be used with RAM, parallel flash, serial flash or mechanical drives
- Support for full speed (12 Mbit/second) and high speed (480 Mbit/second) transfer rates
- OS-abstraction: Can be used with any RTOS, but no OS is required for MSD-only devices

1.3.2.4 How does it work?

Use file system support from host OS

A device which uses emUSB-MSD will be recognized as a mass storage device and can be used like an ordinary disk drive. If the device is unformatted when plugged-in, the host operating system will ask you to format the device. Any file system provided by the host can be used. Typically FAT is used, but other file systems such as NTFS are possible too. If one of those file systems is used, the host is able to read from and write to the device using the storage functions of the emUSB MSD component, which define unstructured read and write operations. Thus, there is no need to develop extra file system code if the application only accesses data on the target from the host side. This is typically the case for simple storage applications, such as USB memory sticks or ATA to USB bridges.

Only provide file system code on the target if necessary

Mass storage devices like USB sticks does not require its own file system implementation. File system program code is only required if the application running on target device has to access the stored data. The development of a file system is a complex

and time-consuming task and enhances the time-to market. Thus we recommend the use of a commercial file system like emFile, Segger's file system for embedded applications. emFile is a high performance library that has been optimized for minimum memory consumption in RAM and ROM, high speed and versatility. It is written in ANSI C and can be used on any CPU and on any media. Refer to www.segger.com/emfile.html for more information about emFile.

1.3.3 emUSB-CDC

emUSB-CDC converts the target device into a serial communication device. A target device running emUSB-CDC is recognized by the host as a serial interface (USB2COM, virtual COM port), without the need to install a special host driver, because the communication device class is one of the standard device classes and every major operating system already provides host drivers for those device classes. All PC software using a COM port will work without modifications with this virtual COM port.

1.3.3.1 Typical applications

Typical applications are:

- Modem
- Telephone system
- Fax machine

1.3.4 emUSB-HID

The Human Interface Device class (HID) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol was defined for the handling of devices which are used by humans to control the operation of computer systems.

An installation of a custom-host USB driver is not necessary, because the USB human interface device class is standardized and every major OS already provides host drivers for it.

1.3.4.1 Typical applications

Typical examples

- Keyboard
- Mouse and similar pointing devices
- Game pad
- Front-panel controls - for example, switches and buttons.
- Bar-code reader
- Thermometer
- Voltmeter
- Low-speed JTAG emulator
- UPS (Uninterruptible power supply)

1.3.5 emUSB-Printer

emUSB-Printer converts the target device into a printing device. A target device running emUSB-Printer is recognized by the host as a printer. Unless the device identifies itself as a printer already recognized by the host PC, you have to install a driver to be able to communicate with the USB device.

1.3.5.1 Typical applications

Typical applications are:

- Laser/Inkjet printer
- CNC machine

1.4 Requirements

1.4.1 Target system

Hardware

The target system must have a USB controller. The memory requirements are approximately 6 Kbytes ROM for the emUSB-Bulk stack or 10 Kbytes ROM for emUSB-Bulk and emUSB-MSD and approximately 1 Kbytes of RAM (only used for buffering). Additionally memory for data storage is required, typically either on-board flash memory (parallel or serial) or an external flash memory card is required. In order to have the control when the device should be enumerated by the host, a switchable attach is necessary. This is a switchable pull-up connected to the D+-Line of USB.

Software

A real-time kernel is required. It can be used with embOS or any compatible RTOS, for information regarding the OS integration refer to *Target OS Interface* on page 261.

1.4.2 Development environment (compiler)

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

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

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

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

1.5 File structure

The following table shows the contents of the emUSB root directory:

Directory	Contents
Application	Contains the application program. Depending on which stack is used, several files are available for each stack. Detailed information can be found in the corresponding chapter.
Config	Contains configuration files (<code>USB_Conf.h</code> , <code>Config_xxx.h</code> , where xxx describes the driver that is used.).
Doc	Contains the emUSB documentation.
Hardware	Contains a simple implementation of the required hardware interface routines. Full implementation of the hardware routine for several CPU and eval board can be found on the SEGGER's website: http://www.segger.com
Inc	Contains include files.
OS	Contains operating systems dependent files which allows to run emUSB with different RTOS's.
USB	Contains the emUSB source code. Note: Do not change the source code in this directory.

Table 1.2: Supplied directory structure of emUSB core package

Depending on the chosen emUSB component, the following additional subdirectories are available:

1.5.1 Bulk communication component

Directory	Contents
Bulk\Windows-Driver	Contains the kernel mode USB driver for the PC (Win32, NT platform), the compiled driver (.sys), the .inf file required for installation. The source code of the Windows driver is included, if a source code version of emUSB-Bulk has been ordered.
Bulk\SampleApp	Contains a PC sample project to help you bring up and test the system.
USB\Bulk	Includes all files that are necessary for the generic bulk communication.

Table 1.3: Additional subdirectories for emUSB bulk communication component

1.5.2 MSD component

Directory	Contents
USB\MSD	Contains all files that handle the specific USB-MSD commands. Different storage drivers, such as a RAM storage device driver or emFile device driver are also available.

Table 1.4: Additional subdirectories for emUSB MSD component

1.5.3 CDC component

Directory	Contents
CDC	The driver installation file (USBser.inf) located in this directory can be used to install the USB-CDC device (Virtual COM-Port) on > Windows 2000 platforms.
USB\CDC	Contains all files specific for the USB-CDC communication.

Table 1.5: Additional subdirectories for emUSB CDC component

1.5.4 HID component

Directory	Contents
HID\SampleApp	Contains a PC sample project to help you bring up and test the system.
USB\HID	Includes all files that are necessary for the HID component.

Table 1.6: Additional subdirectories for emUSB HID communication component

Chapter 2

Background information

This is a short introduction to USB. The fundamentals of USB are explained and links to additional resources are given.
Information provided in this chapter is NOT required to use the software.

2.1 USB

2.1.1 Short Overview

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

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

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

2.1.2 Important USB Standard Versions

USB 1.1 (September 1998)

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

USB 2.0 (April 2000)

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

USB 3.0 (November 2008)

As all previous USB standards, USB 3.0 is fully forward and backward compatible. Existing cables and connectors may be reused but not the new speed can only be used with new USB 3.0 cables and devices. The new speed class is named USB SuperSpeed, which is at a max. rate of 5 GBit/s.

2.1.3 USB System Architecture

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

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

USB Host

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

- A host controller driver (HCD) which provides the functionality of the host controller hardware.
- The USB Driver (USB D) Layer which implements the high level functions used by USB device drivers in terms of the functionality provided by the HCD.
- The USB Device drivers which establish connections to USB devices. The driver classes are also located here and provide generic access to certain types of devices such as printers or mass storage devices.

USB Device

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

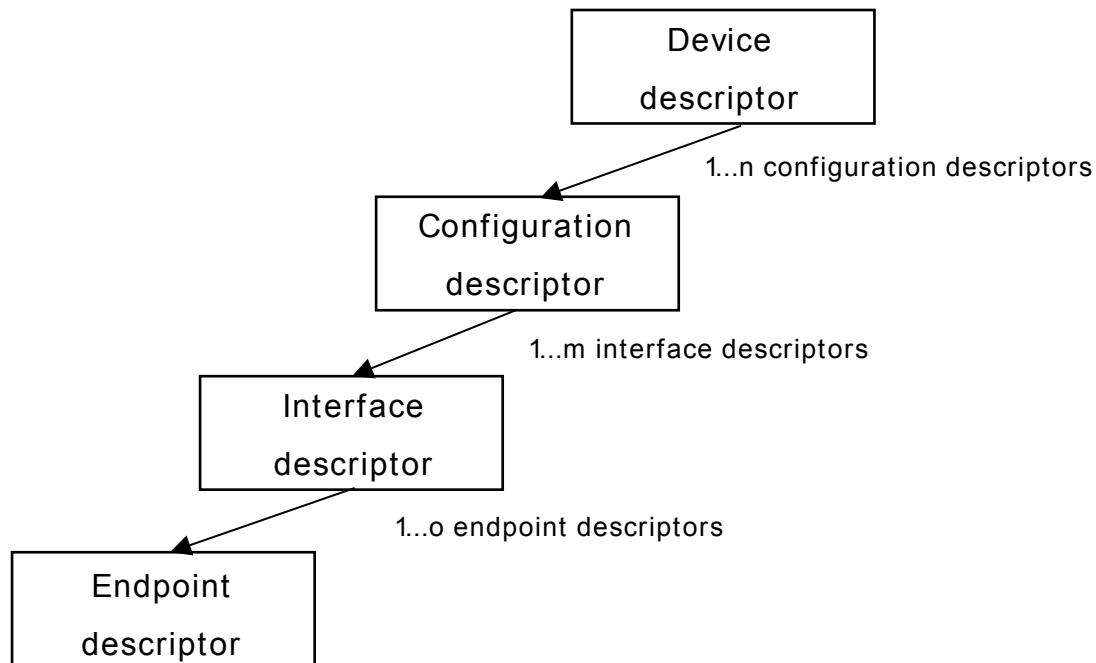
In this guide we treat the term USB device as synonymous with functions and will not consider hubs.

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

Descriptors

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

states the endpoint's address, transfer types etc.



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

2.1.4 Transfer Types

The USB standard defines 4 transfer types: control, isochronous, interrupt, and bulk. Control transfers are used in the setup phase. The application can basically select one of the other 3 transfer types. For most embedded applications, bulk is the best choice because it allows the highest possible data rates.

Control transfers

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

Isochronous transfers

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

Interrupt transfers

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

Bulk transfers

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

2.1.5 Setup phase / Enumeration

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

2.1.6 Product / vendor IDs

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

2.2 Predefined device classes

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

Device classes provide a standardized way of communication between host and device and typically work with a class driver which comes with the host operating system.

Using a predefined device class where applicable minimizes the amount of work to make a device usable on different host systems.

emUSB-Device supports the following device classes:

- Mass Storage Device Class (MSD)
- Human Interface Device Class (HID)
- Communication Device Class (CDC)
- Printer Device Class (PDC)

2.3 USB analyzers

A variety of USB analyzers are on the market with different capabilities.

If you are developing an application using USB, it should not be necessary to have a USB analyzer, but we still recommend you do.

Simple yet powerful USB-Analyzers are available for less than \$1000.

2.4 References

For additional information see the following documents:

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

Chapter 3

Getting started

The first step in getting emUSB up and running is typically to compile it for the target system and to run it in the target system. This chapter explains how to do this.

3.1 How to setup your target system

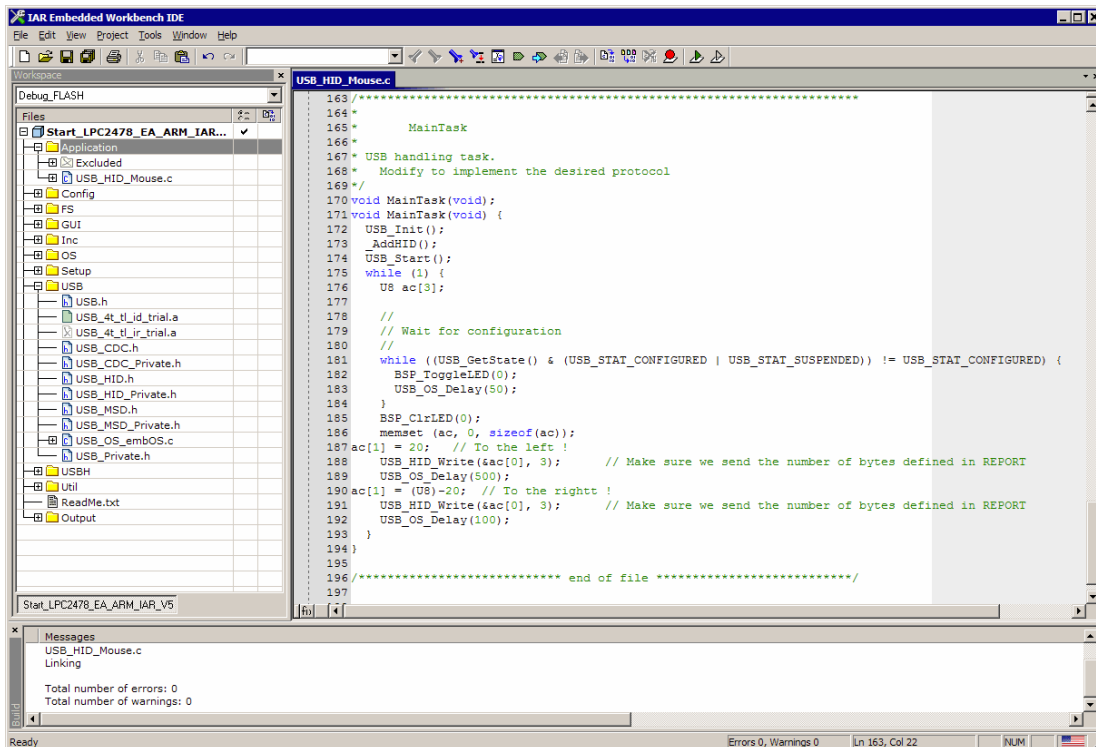
To get the USB up and running, 3 possible ways currently available:

- Upgrade a trial version available on the web with source code.
- Upgrading an embOS Start project
- Creating a project from scratch.

3.1.1 Upgrade a trial version available on the web with source code.

Simply download a trial package available from the SEGGER website.

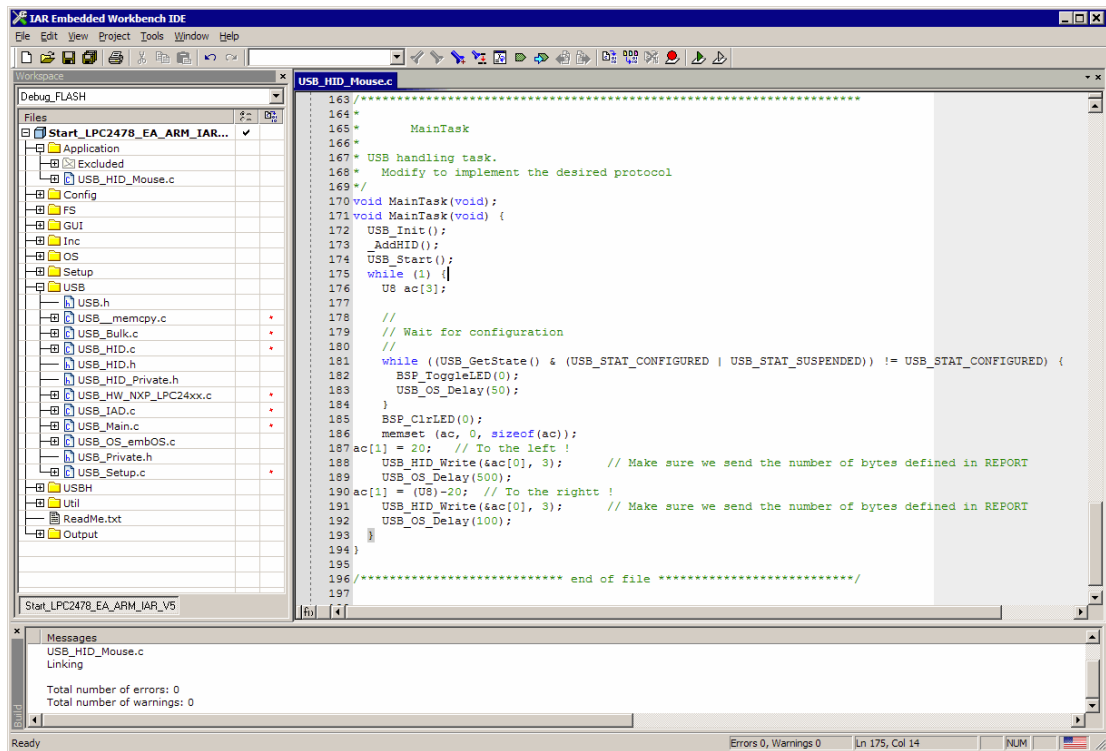
After downloading, extract the trial project and open the workspace/project file which is located in the start folder.



The source files in the USB folder from the emUSB shipment should be copied into

the USB folder of the trial package.

Afterwards the project needs to be updated by adding the source files into the project.



3.1.2 Upgrading an embOS Start project

Integrating emUSB

The emUSB default configuration is preconfigured with valid values, which matches the requirements of the most applications. embUSB 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 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".

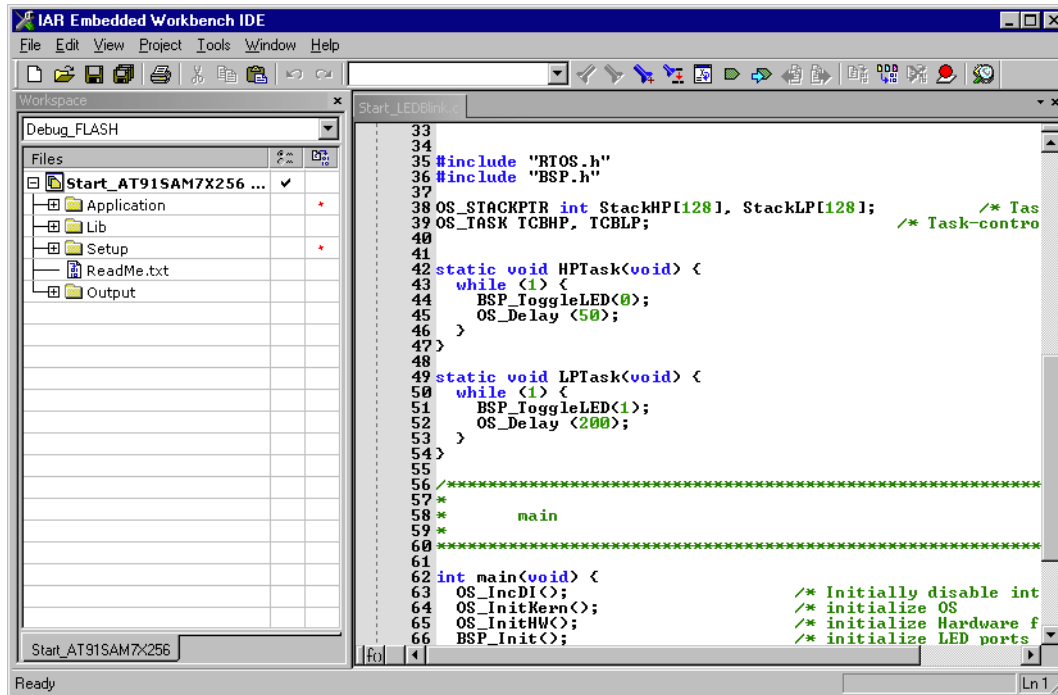
Procedure to follow

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

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

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.



Step 2: Adding emUSB to the start project

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

- Config
- USB
- UTIL (optional)

The `Config` folder includes all configuration files of embUSB. The configuration files are preconfigured with valid values, which match the requirements of most applications. Add the hardware configuration `USB_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 embUSB shipment. It contains optimized MCU and/or compiler specific files, for example a special memcopy func

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
- USB

3.1.3 Creating a project from scratch

To get the target system to behave like a mass storage device or generic bulk device on the USB bus, a few steps have to be taken:

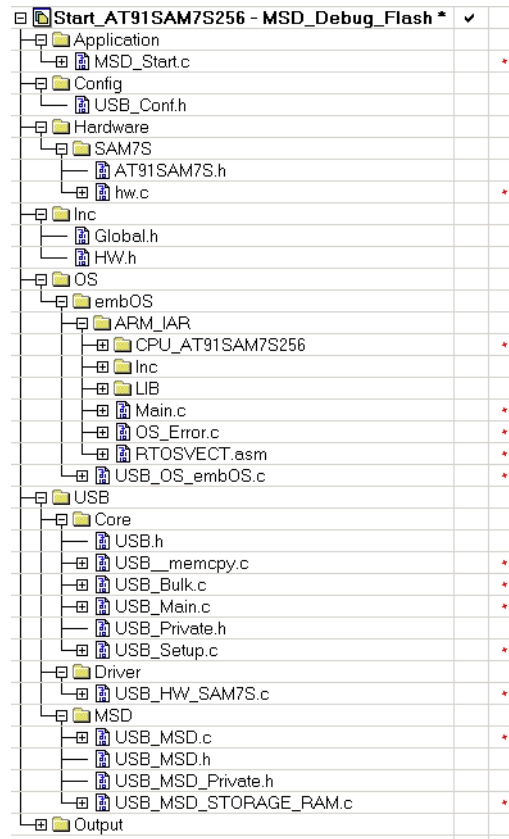
- A project or make file has to be created for the used toolchain.
- The configuration may need to be adjusted.

- The hardware routines for the USB controller have to be implemented.
- Add the path of the required USB header files to the include path.

To get the target up and running is a lot easier if a USB chip is used for which a target hardware driver is already available. In that case, this driver can be used.

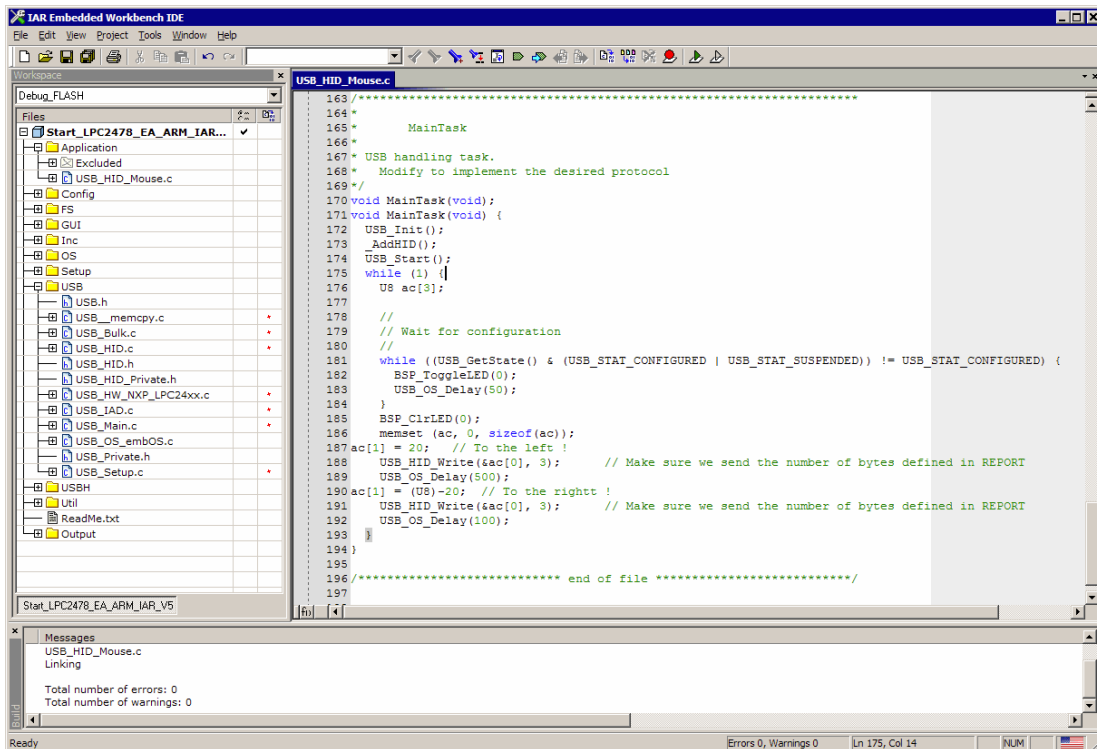
Creating the project or make file

The screenshot below gives an idea about a possible project setup.



3.2 Select the start application

For quick and easy testing of your emUSB integration, start with the code found in the folder Application. Add USB_HID_Mouse.c as your applications to your project.



3.3 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.

After connecting the USB cable to the target device, the mouse pointer should hop from left to right.

3.4 Configuration

An application using emUSB must contain the following functions:

Function	Description
General emUSB configuration functions	
USB_GetVendorId()	Should return the vendor Id of the target.
USB_GetProductId()	Should return the product Id of the target.
USB_GetVendorName()	Should return the manufacturer name.
USB_GetProductName()	Should return the product Id of the target.
USB_GetSerialNumber()	Should return the manufacturer name.
Additional required configuration functions for emUSB-MSD	
USB_MSD_GetVendorName()	Should return the vendor name.
USB_MSD_GetProductVer()	Should return the product version.
USB_MSD_GetSerialNo()	Should return the serial number.

Table 3.1: Functions that are required in emUSB applications

These functions are included in the every example application. The functions could be used without modifications in the development phase of your application, but you may not bring a product on the market without modifying the information like vendor Id and product Id.

Ids	Description
Default vendor Id for all applications	
0x8765	Example vendor Id for all examples.
Used product Ids	
0x1234	Example product Id for all bulk samples.
0x1000	Example product Id for all MSD samples.
0x1200	Example product Id for the MSD CD-ROM sample.
0x1111	Example product Id for all CDC samples.
0x1112	Example product Id for HID mouse sample.
0x1114	Example product Id for the vendor specific HID sample.
0x2114	Example product Id for the Printer class sample.

Table 3.2: List of used product and vendor Ids

3.4.1 General emUSB configuration functions

3.4.1.1 USB_GetVendorId()

Description

Should return the vendor Id of the target.

Prototype

```
U16 USB_GetVendorId(void);
```

Example

```
U16 USB_GetVendorId(void) {  
    return 0x8765;  
}
```

Additional information

The vendor Id is assigned by the USB Implementers forum (www.usb.org). For tests, the default number above (or pretty much any other number) can be used. However, you may not bring a product on the market without having been assigned your own vendor Id.

For emUSB-Bulk and emUSB-CDC: If you change this value, do not forget to make the same change to the `.inf` file as described in section *The .inf file* on page 78 or *The .inf file* on page 183. Otherwise, the Windows host will be unable to locate the driver.

3.4.1.2 USB_GetProductId()

Description

Should return the product Id of the target.

Prototype

```
U16 USB_GetProductId(void);
```

Example

```
U16 USB_GetProductId(void) {  
    return 0x1111;  
}
```

Additional information

The product Id in combination with the vendor Id creates a worldwide unique identifier. For tests, you can use the default number above (or pretty much any other number).

For emUSB-Bulk and emUSB-CDC: If you change this value, do not forget to make the same change to the `.inf` file as described in section *The .inf file* on page 78 or *The .inf file* on page 183. Otherwise, the Windows host will be unable to locate the driver.

3.4.1.3 USB_GetVendorName()

Description

Should return the manufacturer name.

Prototype

```
const char * USB_GetVendorName(void);
```

Example

```
const char * USB_GetVendorName(void) {  
    return "Vendor";  
}
```

Additional information

The manufacturer name is used during the enumeration phase. The product name and the serial number should together give detailed information about which device is connected to the host.

3.4.1.4 USB_GetProductName()

Description

Should return the product name.

Prototype

```
const char * USB_GetProductName(void);
```

Example

```
const char * USB_GetProductName(void) {  
    return "Bulk device";  
}
```

Additional information

The product name is used during the enumeration phase. The manufacturer name and the serial number should together give detailed information about which device is connected to the host.

3.4.1.5 USB_GetSerialNumber()

Description

Should return the serial number.

Prototype

```
const char * USB_GetSerialNumber(void);
```

Example

```
const char * USB_GetSerialNumber(void) {  
    return "12345678";  
}
```

Additional information

The serial number is used during the enumeration phase. The manufacturer and the product name should together give detailed information to the user about which device is connected to the host.

3.4.2 Additional required configuration functions for emUSB-MSD

Refer to *Configuration* on page 141 for more information about the required additional configuration functions for emUSB-MSD.

3.4.3 Descriptors

All configuration descriptors are automatically generated by emUSB and do not require configuration.

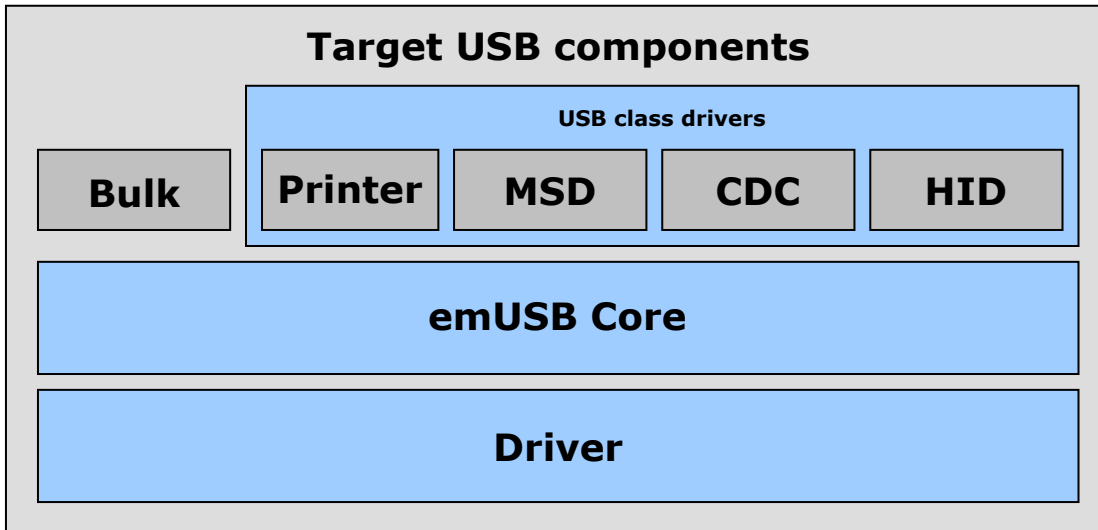
Chapter 4

USB Core

This chapter describes the basic functions of the USB Core.

4.1 Overview

This chapter describes the functions of the core layer of USB Core. These functions are required for all USB class drivers and the unclassified bulk communication component.



General information

To communicate with the host, the example application project includes a USB-specific header `USB.h` and one of the USB libraries (or instead of the libraries the source files, if you have a source version of USB Core). These libraries contain API functions to communicate with the USB host through the USB Core driver.

Every application using USB Core has to perform the following steps:

1. Initialize the USB stack. To initialize the USB stack `USB_Init()` has to be called. `USB_Init()` performs the low-level initialization of the USB stack and calls `USB_X_AddDriver()` to add a driver to the USB stack.
2. Add communication endpoints. You have to add the required endpoints with the compatible transfer type for the desired interface before you can use any of the USB class drivers or the unclassified bulk communication component.
For the emUSB bulk component, refer to `USB_BULK_INIT_DATA` on page 106 for information about the initialization structure that is required when you want to add a bulk interface.
For the emUSB MSD component, refer to `USB_MSD_INIT_DATA` on page 159 and `USB_MSD_INST_DATA` on page 161 for information about the initialization structures that are required when you want to add an MSD interface.
For the emUSB CDC component, refer to `USB_CDC_INIT_DATA` on page 206 for information about the initialization structure that is required when you want to add a CDC interface.
For the emUSB HID component, refer to `USB_HID_INIT_DATA` on page 224 for information about the initialization structure that is required when you want to add a HID interface.
3. Start the USB stack. Call `USB_Start()` to start the USB stack.

Example applications for every supported USB class and the unclassified bulk component are supplied. We recommend to use one of these examples as starting point for your own application. All examples are supplied in the `\Application\` directory.

4.2 Target API

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

Function	Description
USB basic functions	
<code>USB_AddDriver()</code>	Adds a USB device driver to the USB stack.
<code>USB_GetState()</code>	Returns the state of the USB device.
<code>USB_Init()</code>	Initializes USB Core.
<code>USB_IsConfigured()</code>	Checks if the USB device is configured.
<code>USB_Start()</code>	Starts the emUSB core.
USB configuration functions	
<code>USB_AddEP()</code>	Returns an endpoint "handle" that can be used for the desired USB interface.
<code>USB_SetAddFuncDesc()</code>	Sets a callback for setting additional information into the configuration descriptor.
<code>USB_SetClassRequestHook()</code>	Sets a callback to handle class setup requests.
<code>USB_SetVendorRequestHook()</code>	Sets a callback to handle vendor setup requests.
<code>USB_SetIsSelfPowered()</code>	Sets whether the device is self-powered or not.
<code>USB_SetMaxPower()</code>	Sets the target device current consumption.
<code>USB_SetOnRxEP0()</code>	Sets a callback to handle data read of endpoint 0.
<code>USB_SetOnSetupHook()</code>	Sets a callback to handle EP0 setup packets.
<code>USB__WriteEP0FromISR()</code>	Writes data to a USB EP.
<code>USB_StallEP()</code>	Stalls an endpoint.
<code>USB_WaitForEndOfTransfer()</code>	Waits for a data transfer to be completed.
USB IAD functions	
<code>USB_EnableIAD()</code>	Allows to combine multi-interface device classes with single-interface classes.
USB RemoteWakeUp functions	
<code>USB_SetAllowRemoteWakeUp()</code>	Allows the device to publish that remote wake is available.
<code>USB_DoRemoteWakeUp()</code>	Performs a remote wakeup to the host.

Table 4.1: Target USB Core interface function list

4.2.1 USB basic functions

4.2.1.1 USB_AddDriver()

Description

Adds a USB device driver to the USB stack. This function should be called from within `USB_X_AddDriver()` which is implemented in `USB_X.c`.

Prototype

```
void USB_AddDriver(const USB_HW_DRIVER * pDriver);
```

Additional information

To add the driver, use `USB_AddDriver()` with the identifier of the compatible driver. Refer to the section *Available USB drivers* on page 278 for a list of supported devices and their valid identifiers.

Example

```
USB_AddDriver(&USB_Driver_Atme1RM9200);
```

4.2.1.2 USB_GetState()

Description

Returns the state of the USB device.

Prototype

```
int USB_GetState(void);
```

Return value

The return value is a bitwise OR combination of the following state flags.

USB state flags	
USB_STAT_ATTACHED	Device is attached.
USB_STAT_READY	Device is ready.
USB_STAT_ADDRESSED	Device is addressed.
USB_STAT_CONFIGURED	Device is configured.
USB_STAT_SUSPENDED	Device is suspended.

Additional information

A USB device has several possible states. Some of these states are visible to the USB and the host, while others are internal to the USB device. Refer to *Universal Serial Bus Specification*, Revision 2.0, Chapter 9 for detailed information.

4.2.1.3 USB_Init()

Description

Initializes the USB device with its settings.

Prototype

```
void USB_Init(void);
```

4.2.1.4 USB_IsConfigured()

Description

Checks if the USB device is initialized and ready.

Prototype

```
char USB_IsConfigured(void);
```

Return value

0: USB device is not configured.

1: USB device is configured.

4.2.1.5 USB_Start()

Description

Starts USB Core.

Prototype

```
void USB_Start(void);
```

Additional information

This function should be called after configuring USB Core. It initiates a hardware attach and updates the endpoint configuration. When the USB cable is connected to the device, the host will start enumeration of the device.

4.2.2 USB configuration functions

4.2.2.1 USB_AddEP()

Description

Returns an endpoint "handle" that can be used for the desired USB interface.

Prototype

```
unsigned USB_AddEP(U8          InDir,
                  U8          TransferType,
                  U16         Interval,
                  U8          * pBuffer,
                  unsigned     BufferSize);
```

Parameter	Description
InDir	Specifies the direction of the desired endpoint.
TransferType	Specifies the transfer type of the endpoint. The following values are allowed: USB_TRANSFER_TYPE_BULK USB_TRANSFER_TYPE_ISO USB_TRANSFER_TYPE_INT
Interval	Specifies the interval in microframes [0.125 µs] for the endpoint. This value can be zero for a bulk endpoint.
pBuffer	Pointer to a buffer that is used for OUT-transactions. For IN-endpoints this parameter can be NULL.
BufferSize	Size of the buffer.

Table 4.2: USB_AddEP() parameter list

Return value

On success: A valid endpoint handle is returned.
On failure: 0 is returned.

4.2.2.2 USB_SetAddFuncDesc()

Description

Sets a callback for setting additional information into the configuration descriptor.

Prototype

```
void USB_SetAddFuncDesc(USB_ADD_FUNC_DESC * pAddDescFunc);
```

Parameter	Description
<code>pfAddDescFunc</code>	Pointer to a function that should be called when building the configuration descriptor.

Table 4.3: USB_SetAddFuncDesc() parameter list

Additional information

USB_ADD_FUNC_DESC is defined as follows:

```
typedef void USB_ADD_FUNC_DESC(USB_INFO_BUFFER * pInfoBuffer);
```

4.2.2.3 USB_SetClassRequestHook()

Description

Sets a callback for a function that handles setup class request packets.

Prototype

```
void USB_SetClassRequestHook(unsigned Interface,
                             USB_ON_CLASS_REQUEST * pfOnClassrequest);
```

Parameter	Description
<code>Interface</code>	Specifies the Interface number of the class on which the hook will be installed.
<code>pfOnClassrequest</code>	Pointer to a function that should be called when a setup class request/packet is received.

Table 4.4: USB_SetClassRequestHook() parameter list

Additional information

Note that the callback will be called within an ISR.

If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.

`USB_ON_CLASS_REQUEST` is defined as follows:

```
typedef void USB_ON_CLASS_REQUEST(const USB_SETUP_PACKET * pSetup-
Packet);
```

4.2.2.4 USB_SetVendorRequestHook()

Description

Sets a callback for a function that handles setup vendor request packets.

Prototype

```
void USB_SetClassRequestHook(unsigned Interface,
                             USB_ON_CLASS_REQUEST * pfOnClassrequest);
```

Parameter	Description
Interface	Specifies the Interface number of the class on which the hook will be installed.
pfOnClassrequest	Pointer to a function that should be called when a setup vendor request/packet is received.

Table 4.5: USB_SetClassRequestHook() parameter list

Additional information

Note that the callback will be called within an ISR.

If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.

`USB_ON_CLASS_REQUEST` is defined as follows:

```
typedef void USB_ON_CLASS_REQUEST(const USB_SETUP_PACKET * pSetup-
Packet);
```

4.2.2.5 USB_SetIsSelfPowered()

Description

Sets whether the device is self-powered or not.

Prototype

```
void USB_SetIsSelfPowered(U8 IsSelfPowered);
```

Parameter	Description
<code>IsSelfPowered</code>	0 - Device is not self-powered. 1 - Device is self-powered..

Table 4.6: USB_SetClassRequestHook() parameter list

Additional information

This function should be called before `USB_Start()`, as it will specify if the device is self-powered or not.

The default value is 0 (not self-powered).

4.2.2.6 USB_SetMaxPower()

Description

Sets the max power consumption that the target should report during enumeration.

Prototype

```
void USB_SetMaxPower(unsigned MaxPower);
```

Parameter	Description
MaxPower	Specifies the max power consumption given in mA. MaxPower should be in range between 0mA - 500mA.

Table 4.7: USB_SetClassRequestHook() parameter list

Additional information

This function should be called before `USB_Start()`, as it will specify how much power the device will consume from the host.

The default value is 100mA.

4.2.2.7 USB_SetOnRxEP0()

Description

Sets a callback to handle data read of endpoint 0.

Prototype

```
void USB_SetOnRxEP0(USB_ON_RX_FUNC * pfOnRx);
```

Parameter	Description
pfOnRx	Pointer to a function that should be called when receiving data other than setup packets.

Table 4.8: USB_SetOnRxEP0() parameter list

Additional information

Note that the callback will be called within an ISR.

If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.

`USB_ON_RX_FUNC` is defined as follows:

```
typedef void USB_ON_RX_FUNC(const U8 * pData, unsigned NumBytes);
```

4.2.2.8 USB_SetOnSetupHook()

Description

Sets a callback for a function that handles setup class request packets.

Prototype

```
void USB_SetClassRequestHook(unsigned Interface,
                             USB_ON_CLASS_REQUEST * pfOnClassrequest);
```

Parameter	Description
Interface	Specifies the Interface number of the class on which the hook will be installed.
pfOnClassrequest	Pointer to a function that should be called when a setup class request/packet is received.

Table 4.9: USB_SetClassRequestHook() parameter list

Additional information

Note that the callback will be called within an ISR.

If it is necessary to send data from the callback function through endpoint 0, use the function `USB__WriteEP0FromISR()`.

`USB_ON_CLASS_REQUEST` is defined as follows:

```
typedef int USB_ON_SETUP(const USB_SETUP_PACKET * pSetupPacket);
```

4.2.2.9 USB__WriteEP0FromISR()

Description

Writes data to a USB EP.

Prototype

```
void USB__WriteEP0FromISR(const void* pData, unsigned NumBytes,
                           char Send0PacketIfRequired)
```

Parameter	Description
<code>pData</code>	Data that should be written.
<code>NumBytes</code>	Number of bytes to write.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of <code>MaxPacketSize</code> . Normally <code>MaxPacketSize</code> for control mode transfer is 64 byte.

Table 4.10: USB__WriteEP0FromISR() parameter list

4.2.3 USB control functions

4.2.3.1 USB_StallEP()

Description

Stalls an endpoint.

Prototype

```
void USB_StallEP(U8 EPIndex);
```

Parameter	Description
EPIndex	Endpoint handle that needs to be stalled.

Table 4.11: USB_StallEP() parameter list

4.2.3.2 USB_WaitForEndOfTransfer()

Description

Waits for a data transfer to be completed.

Prototype

```
void USB_WaitForEndOfTransfer(U8 EPIndex);
```

Parameter	Description
EPIndex	Endpoint handle to wait for end of transfer.

Table 4.12: USB_WaitForEndOfTransfer() parameter list

4.2.4 USB IAD functions

4.2.4.1 USB_EnableIAD()

Description

Allows to combine multi-interface device classes with single-interface classes or other multi-interface classes.

Prototype

```
void USB_EnableIAD(void);
```

Additional information

Simple device classes such as HID and MSD or BULK use only one interface descriptor to describe the class. The interface descriptor also contains the device class code. The CDC device classes uses more than one interface descriptor to describe the class. The device class code will then be written into the device descriptor. It may be possible to add an interface which does not belong to the CDC class, but it may not be correctly recognized by the host.

In order to allow this, a new descriptor type was introduced:

IAD (Interface Association Descriptor), this descriptor will encapsulate the multi-interface class into this IA descriptor, so that it will be seen as one single interface and will then allow to add other device classes.

If you intend to use the CDC component with any other component, please call `USB_EnableIAD()` before adding the CDC component through `USB_CDC_Add()`.

4.2.5 USB Remote wakeup functions

Remote wakeup is a feature that allows a device to wake up a host system from a suspend state.

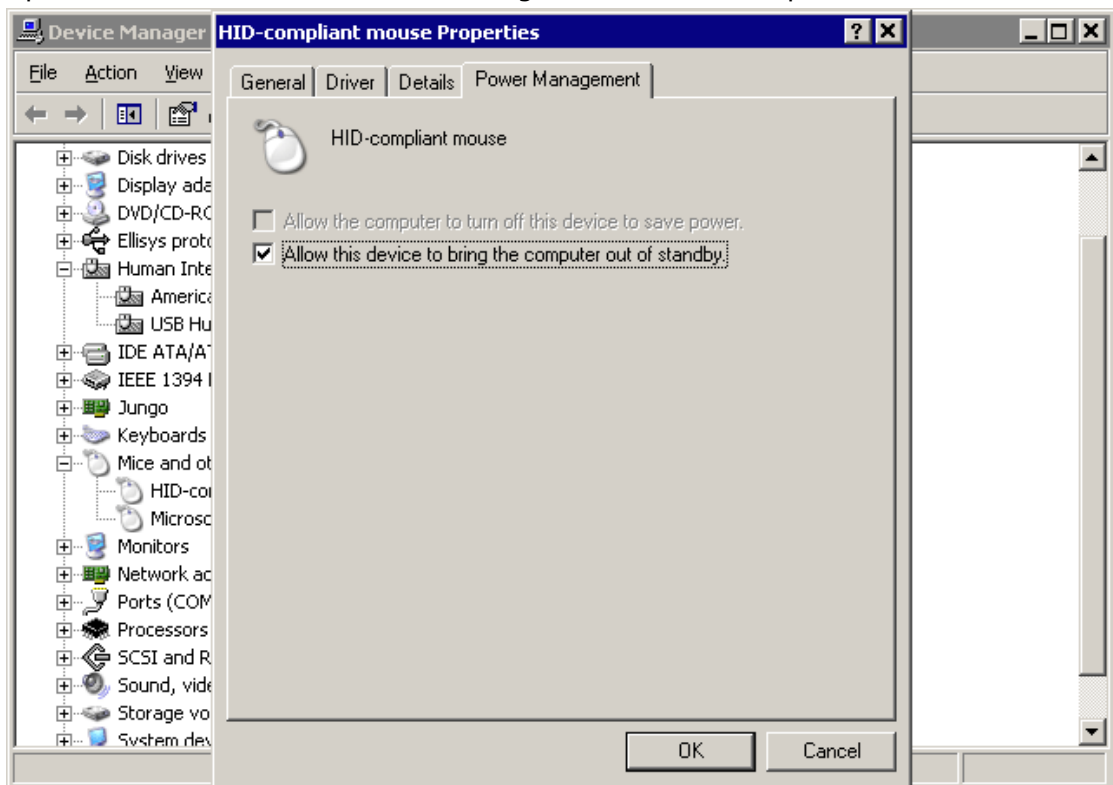
In order to do this a special resume signal is sent over the USB data lines. This signal should be held for at least 1ms but not more than 15 ms. Typically this signaling is held for 10ms.

Additionally the USB host controller and operating system should be able to handle this signaling.

Windows OS:

Currently Windows OS only supports the wakeup feature on device are based on HID mouse/keyboard, CDC Modem and RNDIS ethernet class. MSD, generic bulk and CDC serial is not supported by Windows. So therefore a HID mouse class even as dummy interface within you USB configuration is currently mandatory. A sample is provided for adding such a dummy class.

Windows must also be told that the device should wake the PC from the suspend state. This is done by setting the option "Allow this device to bring the computer out of standby.". This is done by opening the device manager, then the device properties of the device (in most cases this device is called HID-compliant mouse) should be opened and within the "Power Management" the said option should be checked.



MAC OS X

MAC OS X supports remote wakeup for all device classes.

4.2.5.1 USB_SetAllowRemoteWakeUp()

Description

Allows the device to publish that remote wake is available.

Prototype

```
void USB_SetAllowRemoteWakeUp(U8 AllowRemoteWakeup);
```

Parameter	Description
AllowRemoteWakeup	1 - Allows and publish the remote wakeup is available. 0 - Publish that remote wakeup is not available.

Table 4.13: USB_SetAllowRemoteWakeUp() parameter list

Additional information

This function should be called before the function USB_Start() is called. This make sure that the Host is informed that USB remote wake up is available.

4.2.5.2 USB_DoRemoteWakeup()

Description

Performs a remote wakeup in order to wake up the host from the standby/suspend state.

Prototype

```
void USB_DoRemoteWakeUp(void);
```

Additional information

Please make sure that this function is only called within a task. The reason for this is that USB_DoRemoteWakeup uses functions which are not allowed to be called within an ISR routine.

Chapter 5

Bulk communication

This chapter describes how to get the emUSB-Bulk up and running.

5.1 Generic bulk stack

The generic bulk stack is located in the directory `USB`. All C files in the directory should be included in the project (compiled and linked as part of your project). The files in this directory are maintained by SEGGER and should not require any modification. All files requiring modifications have been placed in other directories.

5.2 The Kernel mode driver (PC)

In order to communicate with a target (client) running emUSB, an emUSB bulk kernel mode driver has to be installed on Windows PC's. Typically, this is done as soon as emUSB runs on target hardware.

Installation of the driver as well as how to recompile it is explained in this chapter.

5.2.1 Why is a driver necessary?

In Microsoft's Windows operating systems, all communication with real hardware is implemented with *kernel-mode* drivers. Normal applications run in *user-mode*. In user-mode, hardware access is not permitted. All access to hardware is done through the operating system. The operating system uses a kernel mode driver to access the actual hardware. In other words: every piece of hardware requires one or more kernel mode drivers to function. Windows supplies drivers for most common types of hardware, but it does not come with a generic bulk communication driver. It comes with drivers for certain classes of devices, such as keyboard, mouse and mass storage device (for example, a USB stick). This makes it possible to connect a USB mouse and not having to install a driver for it: Windows already has a driver for it.

Unfortunately, there is no generic kernel mode driver which allows communication to any type of device in bulk mode. This is why a kernel mode driver needs to be supplied in order to work with emUSB-Bulk.

5.2.2 Supported platforms

The kernel mode driver works on all NT-type platforms. This includes Windows 2000 and Windows XP (home and professional), Windows 2003 Server, Windows Vista. Windows NT itself does not support USB; Win98 is not supported by the driver.

5.3 Installing the driver

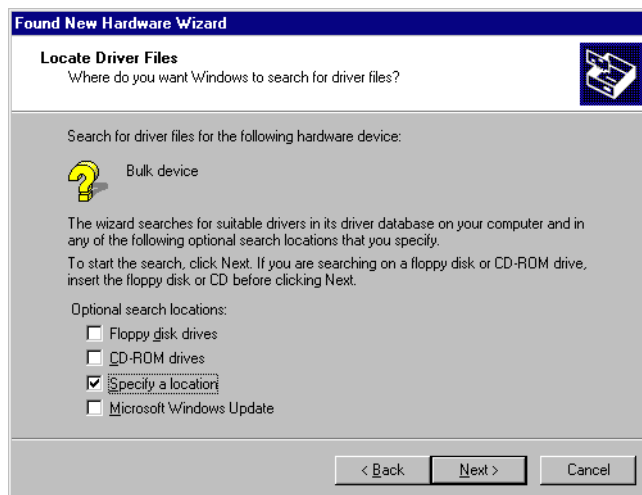
When the target device is plugged on the computer's USB port, or when the computer is first powered up after connecting the emUSB device, Windows will detect the new hardware.



The wizard will complete the installation for the detected device. First select the **Search for a suitable driver for my device** option and click on the **Next** button.



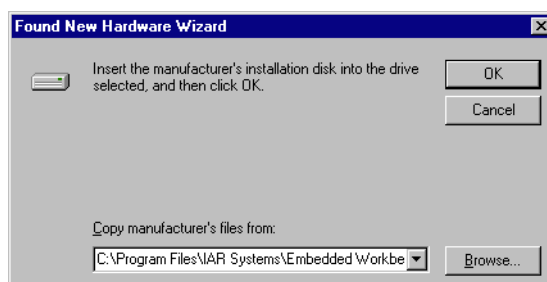
In the next step, select the **Specify a location** option and click afterwards on the **Next** button.



The wizard needs the path to the correct driver files for the new device.



Use the directory navigator to select the `USBBulk.inf` file and click the **Open** button.



The wizard confirms the choice and starts to copy, after clicking the **Next** button.



At this point, the installation is complete. Click the **Finish** button to dismiss the wizard.

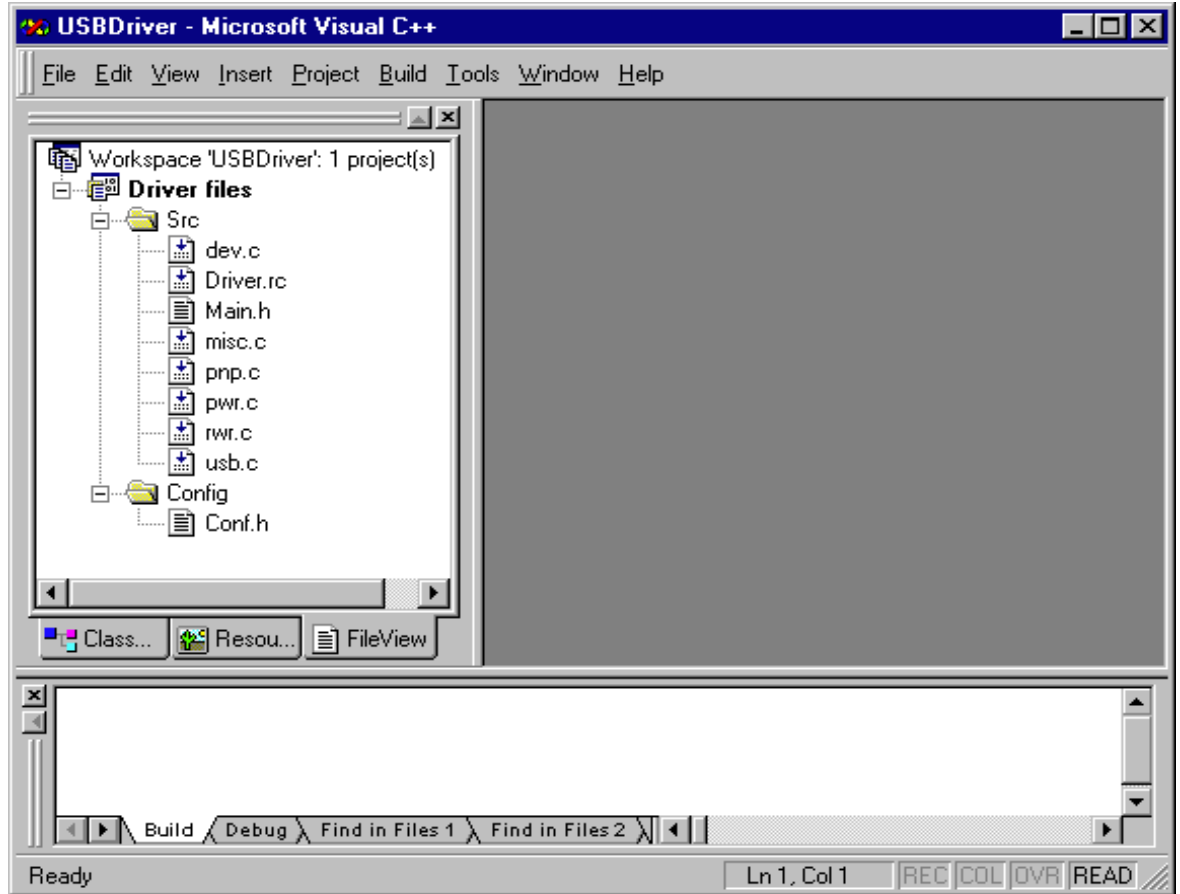


5.3.1 Recompiling the driver

To recompile the driver, the Device Developer Kit (NTDDK), as well as an installation of Microsoft Visual C++ 6.0 or Visual Studio .net is needed.

The workspace is placed in the subdirectory `Driver`. In order to open it, double click the workspace file `USBDriver.dsw`.

A workspace similar to the screenshot below is opened.



Choose **Build | Build USBBulk.sys** (Shortcut: F7) to compile and link the driver.

5.3.2 The .inf file

The .inf file is required for installation of the kernel mode driver.

It looks as follows:

```

;
;
;       USB BULK Device driver inf
;
;
[Version]
Signature="$CHICAGO$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
provider=%MfgName%
DriverVer=08/07/2003

[SourceDisksNames]
1="USB BULK Installation Disk",,,

[SourceDisksFiles]
USBBulk.sys = 1
USBBulk.inf = 1

[Manufacturer]
%MfgName%=DeviceList

[DeviceList]
%USB\VID_8765&PID_1234.DeviceDesc%=USBBULK.Dev, USB\VID_8765&PID_1234

; [PreCopySection]
; HKR,,NoSetupUI,,1

[DestinationDirs]
USBBULK.Files.Ext = 10,System32\Drivers

[USBBULK.Dev]
CopyFiles=USBBULK.Files.Ext
AddReg=USBBULK.AddReg

[USBBULK.Dev.NT]
CopyFiles=USBBULK.Files.Ext
AddReg=USBBULK.AddReg

[USBBULK.Dev.NT.Services]
Addservice = USBBULK, 0x00000002, USBBULK.AddService

[USBBULK.AddService]
DisplayName     = %USBBULK.SvcDesc%
ServiceType     = 1                ; SERVICE_KERNEL_DRIVER
StartType       = 3                ; SERVICE_DEMAND_START
ErrorControl    = 1                ; SERVICE_ERROR_NORMAL
ServiceBinary   = %10%\System32\Drivers\USBBULK.sys
LoadOrderGroup = Base

[USBBULK.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,USBBULK.sys

[USBBULK.Files.Ext]
USBBulk.sys

;-----;

```

```
[Strings]
MfgName="MyCompany"
USB\VID_8765&PID_1234.DeviceDesc="USB Bulk Device"
USBBULK.SvcDesc="USB Bulk device driver"
```

red - required modifications

green - possible modifications

You have to personalize the `.inf` file on the red marked positions. Changes on the green marked positions are optional and not necessary for the correct function of the device.

Replace the red marked positions with the personal vendor Id (VID) and product Id (PID). These changes have to be identical with the modifications in the configuration functions to work correct.

The required modifications of the configuration functions are described in the section *Configuration* on page 41.

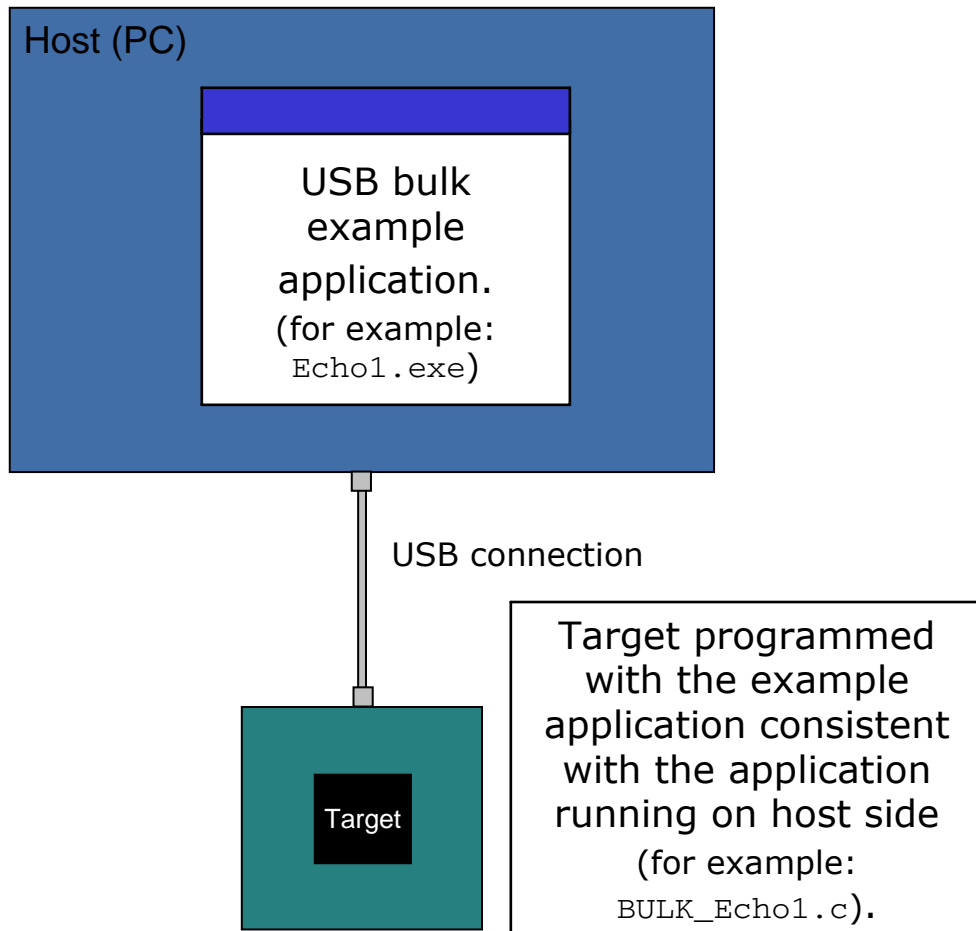
5.3.3 Configuration

To get emUSB up and running as well as doing an initial test, the configuration as it is delivered should not be modified. The configuration section can later on be modified to match your real application. The configuration must only be modified if emUSB should be used in a final product. Refer to section *Configuration* on page 41 to get detailed information about the functions which has to be adapted.

5.4 Example application

Example applications for both the target (client) and the PC (host) are supplied. These can be used for testing the correct installation and proper function of the device running emUSB.

The application is a modified echo server (`BULK_Echo1.c`); the application receives data byte by byte, increments every single byte and sends it back to the host.



To use this application, make sure to use the corresponding example files both on the host-side as on the target side. The example applications on the PC host are named in the same way, just without the prefix `BULK_`. (For example, if the host runs `Echo1.exe`, `BULK_Echo1.c` has to be included into your project, compiled and downloaded into your target.) There are additional examples that can be used for testing emUSB.

The following start application files are provided:

File	Description
<code>BULK_Echo1.c</code>	This application was described in the upper text.
<code>BULK_EchoFast.c</code>	This is the faster version of <code>Bulk_Echo1.c</code>
<code>BULK_Test.c</code>	This application can be used to test emUSB-Bulk with different packet sizes received from and sent to the PC host.

Table 5.1: Supplied sample applications

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

Depending on which application is running on the emUSB device, use one of the following example applications:

File	Description
Echo1.exe	If the <code>BULK_Echo1.c</code> sample application is running on the emUSB-Bulk device, use this application.
EchoFast.exe	If the <code>BULK_EchoFast.c</code> sample application is running on the emUSB-Bulk device, use this <code>EchoFast</code> application.
Test.exe	If the <code>BULK_Test.c</code> application is running on the emUSB-Bulk device, use this application to test the emUSB-Bulk stack.

Table 5.2: Supplied host applications

To use these examples, the application on the PC host should use the same example file to work correctly. The example applications on the PC host are named in the same way. The example applications for the host-side are supplied in both source code and executable form in the `Bulk\SampleApp` directory. For information how to compile the host examples refer to *Compiling the PC example application* on page 83.

The start application will of course later on be replaced by the real application program. For the purpose of getting emUSB up and running as well as doing an initial test, the start application should not be modified.

5.4.1 Running the example applications

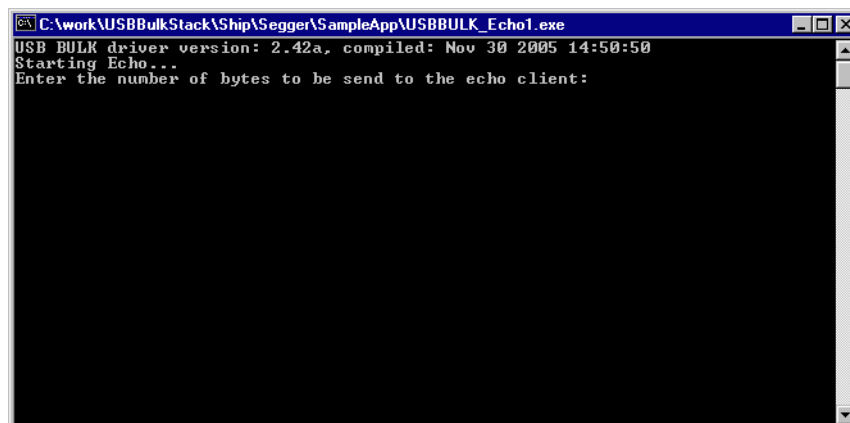
To test the emUSB-Bulk component, build and download the application of choice for the target-side. If you connect your target to the host via USB while the example application is running, Windows will detect the new hardware.

To run one of the example applications, simply start the executable, for example by double clicking it. If the USB-Bulk device is not connected to the PC or the driver is not installed, the following message box should pop up.

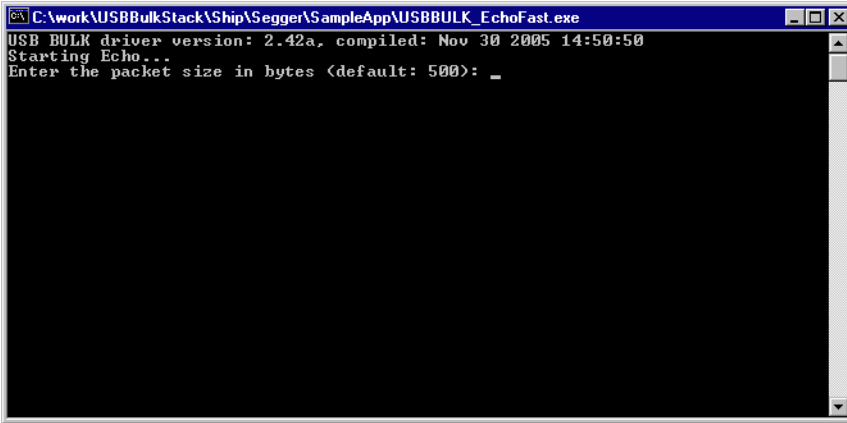


If a connection can be established, it exchanges data with the target, testing the USB connection.

Example output of `Echo1.exe`:



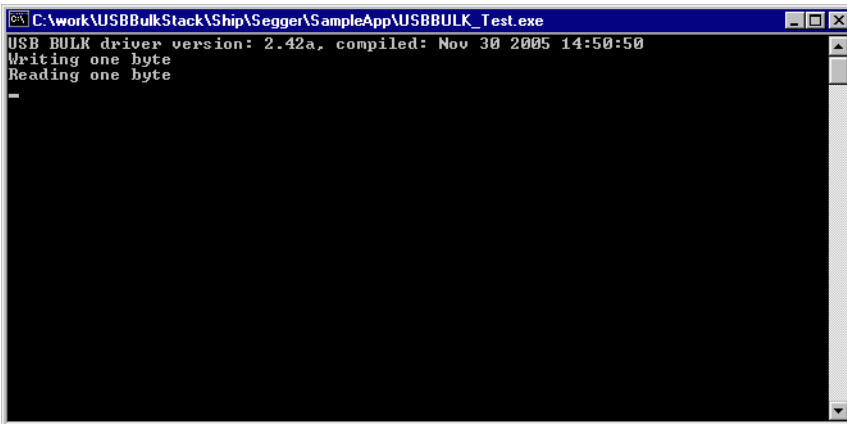
Example output of `EchoFast.exe`:



```

C:\work\USBBulkStack\Ship\Segger\SampleApp\USBBULK_EchoFast.exe
USB BULK driver version: 2.42a, compiled: Nov 30 2005 14:50:50
Starting Echo...
Enter the packet size in bytes (default: 500): _
  
```

Example output of `Test.exe`:



```

C:\work\USBBulkStack\Ship\Segger\SampleApp\USBBULK_Test.exe
USB BULK driver version: 2.42a, compiled: Nov 30 2005 14:50:50
Writing one byte
Reading one byte
_
  
```

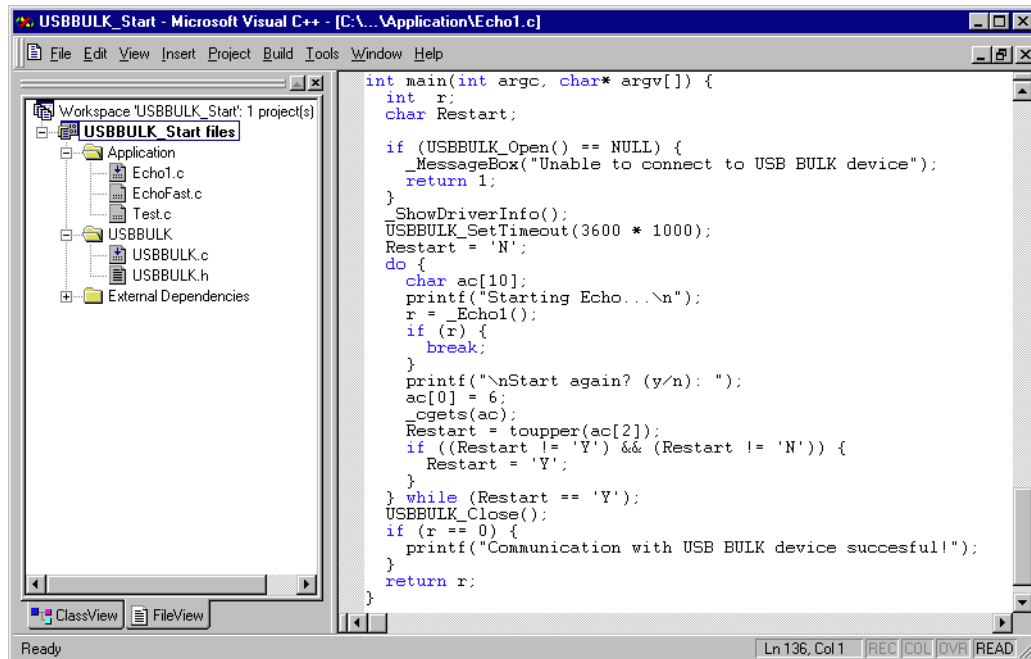
If the host example application can communicate with the emUSB device, the example application will be in interactive mode for the `Echo1` and the `EchoFast` application. In case of an error, a message box is displayed.

Error Messages	Description
Unable to connect to USB BULK device	The USB device is not connected to the PC or the connection is faulty.
Could not write to device	The PC sample application was not able to write one byte.
Could not read from device (time out)	The PC sample application was not able to read one byte.
Wrong data read	The result of the target sample application is not correct.

Table 5.3: List of error messages

5.4.2 Compiling the PC example application

For compiling the example application you need a Microsoft compiler. The compiler is part of Microsoft Visual C++ 6.0 or Microsoft Visual Studio .Net.



```

int main(int argc, char* argv[]) {
    int r;
    char Restart;

    if (USBBULK_Open() == NULL) {
        _MessageBox("Unable to connect to USB BULK device");
        return 1;
    }
    ShowDriverInfo();
    USBBULK_SetTimeout(3600 * 1000);
    Restart = 'N';
    do {
        char ac[10];
        printf("Starting Echo...\n");
        r = _Echo1();
        if (r) {
            break;
        }
        printf("\nStart again? (y/n): ");
        ac[0] = 6;
        _cgets(ac);
        Restart = toupper(ac[2]);
        if ((Restart != 'Y') && (Restart != 'N')) {
            Restart = 'V';
        }
    } while (Restart == 'Y');
    USBBULK_Close();
    if (r == 0) {
        printf("Communication with USB BULK device succesful!");
    }
    return r;
}

```

The source code of the sample application is located in the subfolder `Bulk\SAMPLEAPP`. Open the file `USBBULK_Start.dsw` and compile the source choose **Build | Build SampleApp.exe** (Shortcut: F7). To run the executable choose **Build | Execute SampleApp.exe** (Shortcut: CTRL-F5).

5.5 Target API

This chapter describes the functions that can be used with the target system.

General information

To communicate with the host, the sample application project includes USB-specific header and source files (USB.h, USB_Main.c, USB_Setup.c, USB_Bulk.c, USB_Private.h). These files contain API functions to communicate with the USB host through the emUSB driver.

Purpose of the USB Device API functions

To have an easy start up when writing an application on the device side, these API functions have a simple interface and handle all operations that need to be done to communicate with the host emUSB kernel mode driver.

Therefore, all operations that need to write to or read from the emUSB are handled internally by the provided API functions.

5.5.1 Target interface function list

Routine	Explanation
USB-Bulk functions	
<code>USB_BULK_Add()</code>	Adds an USB-Bulk interface to emUSB.
<code>USB_BULK_CancelRead()</code>	Cancels a non-blocking read operation that is pending.
<code>USB_BULK_CancelWrite()</code>	Cancels a non-blocking write operation that is pending.
<code>USB_BULK_GetNumBytesInBuffer()</code>	Returns the number of byte in BULK-OUT buffer.
<code>USB_BULK_GetNumBytesRemToRead()</code>	Returns the number of bytes which have to be read.
<code>USB_BULK_GetNumBytesToWrite()</code>	Returns the number of bytes which have to be written.
<code>USB_BULK_Read()</code>	USB-Bulk read.
<code>USB_BULK_ReadOverlapped()</code>	Non-blocking version of <code>USB_BULK_Read()</code> .
<code>USB_BULK_ReadTimed()</code>	Starts a read operation that should be done within a given time-out.
<code>USB_BULK_Receive()</code>	Read data from host and return immediately as soon as data has been received.
<code>USB_BULK_SetOnRXHook()</code>	Installs a hook that will be called when an USB packet is received.
<code>USB_BULK_WaitForTX()</code>	Waits for a non-blocking write operation that is pending.
<code>USB_BULK_WaitForRX()</code>	Waits for a non-blocking write operation that is pending.
<code>USB_BULK_Write()</code>	Starts a blocking write operation.
<code>USB_BULK_WriteEx()</code>	Starts a blocking write operation that allows to specify whether a NULL packet should be sent or not.
<code>USB_BULK_WriteExTimed()</code>	Starts an USB-Bulk WriteEx operation that should be done within a given time-out.
<code>USB_BULK_WriteOverlapped()</code>	Non-blocking version of <code>USB_Bulk_Write()</code> .
<code>USB_BULK_WriteOverlappedEx()</code>	Write data to the host asynchronously.
<code>USB_BULK_WriteNULLPacket()</code>	Sends a NULL (zero-length) packet to host.
<code>USB_BULK_WriteTimed()</code>	Starts an USB-Bulk Write operation that should be done within a given time-out.
Data structures	
<code>USB_BULK_INIT_DATA</code>	Initialization structure which is required when adding a bulk interface.
<code>USB_ON_RX_FUNC</code>	Function called when data is received.

Table 5.4: Target interface function list

5.5.2 USB-Bulk functions

5.5.2.1 USB_BULK_Add()

Description

Adds interface for USB-Bulk communication to emUSB.

Prototype

```
void USB_BULK_Add( const USB_BULK_INIT_DATA * pInitData );
```

Parameter	Description
pInitData	Pointer to USB_BULK_INIT_DATA structure.

Table 5.5: USB_BULK_Add() parameter list

Additional information

USB_BULK_INIT_DATA is defined as follows:

```
typedef struct {  
    U8 EPIn;    // Endpoint for sending data to the host  
    U8 EPOut;   // Endpoint for receiving data from the host  
};
```

5.5.2.2 USB_BULK_CancelRead()

Description

Cancels any non-blocking/blocking read operation that is pending.

Prototype

```
void USB_BULK_CancelRead(void);
```

Additional information

This function should be called when a pending asynchronous read operation should be canceled. The function can be called from any task. In case of canceling a blocking operation, this function must be called from another task.

5.5.2.3 USB_BULK_CancelWrite()

Description

Cancels a non-blocking/blocking read operation that is pending.

Prototype

```
void USB_BULK_CancelWrite(void);
```

Additional information

This function should be called when a pending asynchronously write operation should be canceled. It can be called from any task. In case of canceling a blocking operation, this function must be called from another task.

5.5.2.4 USB_BULK_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.

Prototype

```
unsigned USB_BULK_GetNumBytesInBuffer(void);
```

5.5.2.5 USB_BULK_GetNumBytesRemToRead()

Description

Returns the remaining number of bytes to read.

Prototype

```
unsigned USB_BULK_GetNumBytesRemToRead(void);
```

5.5.2.6 USB_BULK_GetNumBytesToWrite()

Description

Returns the number of bytes that should be written.

Prototype

```
unsigned USB_BULK_GetNumBytesToWrite(void);
```

5.5.2.7 USB_BULK_Read()

Description

Reads data from the host.

Prototype

```
int USB_BULK_Read(void* pData, unsigned NumBytes);
```

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 5.6: USB_BULK_Read() parameter list

Return value

Number of bytes that have been received.
In case of an error -1 is returned.

5.5.2.8 USB_BULK_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USB_BULK_ReadOverlapped(void* pData, unsigned NumBytes);
```

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 5.7: USB_BULK_ReadOverlapped() parameter list

Return value

Number of bytes that have already been received or have been copied from internal buffer.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, `USB_BULK_WaitForRX()` needs to be called.

5.5.2.9 USB_BULK_ReadTimed()

Description

Reads data from the host with a given time-out.

Prototype

```
int USB_BULK_ReadOverlapped(void* pData, unsigned NumBytes, unsigned ms);
```

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Time-out given in milliseconds.

Table 5.8: USB_BULK_ReadTimed() parameter list

Return value

Number of bytes that have been read within the given time-out.

Additional information

This function blocks a task until all data have been read or a time-out occurs. This function also returns when target is disconnected from host or when a USB reset occurs.

5.5.2.10 USB_BULK_SetOnRXHook()

Description

Sets a callback that can be set whenever a data packet was received from the host.

Prototype

```
void USB_BULK_SetOnRXHook(USB_ON_RX_EP * pfOnRx);
```

Parameter	Description
pfOnRx	Pointer to the callback function.

Table 5.9: USB_BULK_SetOnRXHook() parameter list

Additional information

This function can be used to set up a monitoring task which would suspend the calling task and resume it when data is received.

The callback function will be called within a interrupt service routine, so minimal operations should be done within this function.

5.5.2.11 USB_BULK_Receive()

Description

Reads data from host and returns as soon as data has been received.

Prototype

```
int USB_BULK_Receive(void * pData, unsigned NumBytes);
```

Parameter	Description
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.

Table 5.10: USB_BULK_Receive() parameter list

Return value

Number of bytes that have been read.

Additional information

If no error occurs, this function returns the number of bytes received.

In case of an error, -1 is returned.

Calling `USB_BULK_Receive()` will return as much data as is currently available—up to the size of the buffer specified.

5.5.2.12 USB_BULK_WaitForRX()

Description

Waits for reading data transfer from the host to be completed.

Prototype

```
void USB_BULK_WaitForRX(void);
```

Additional information

This function should be called in order to synchronize the task with the read data transfer that previously initiated.

5.5.2.13 USB_BULK_WaitForTX()

Description

Waits for writing data transfer to the host to be completed.

Prototype

```
void USB_BULK_WaitForTX(void);
```

5.5.2.14 USB_BULK_Write()

Description

Sends data to the USB host.

Prototype

```
int USB_BULK_Write(const void * pData, unsigned NumBytes);
```

Parameter	Description
pData	Data that should be written.
NumBytes	Number of bytes to write.

Table 5.11: USB_BULK_Write() parameter list

Return value

Number of bytes that have been written.

5.5.2.15 USB_BULK_WriteEx()

Description

Sends data to the host with the option to send a zero-length packet at the end of the data transfer.

Prototype

```
int USB_BULK_WriteEx(const void* pData,
                    unsigned   NumBytes,
                    char       Send0PacketIfRequired);
```

Parameter	Description
<code>pData</code>	Pointer to a buffer that contains the written data.
<code>NumBytes</code>	Number of bytes to write.
<code>Send0PacketIfRequired</code>	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of <code>MaxPacketSize</code> . Normally <code>MaxPacketSize</code> for full-speed devices is 64 byte. For high-speed devices the normal packet size is between 64-512 bytes.

Table 5.12: USB_BULK_WriteEx() parameter list

Additional information

Normally `USB_BULK_Write` is called to let the stack send that whole packet to the host and send an optional zero-length packet to tell the host that this was the last packet. This is the case when the last packet that should be sent is `MaxPacketSize` long.

When using this function, the zero-length packet handling can be controlled. This means the function can be called when sending, data should be sent in multiple steps. Please make sure that `NumBytes` is always except for the last transmission, a multiple of `MaxPacketSize`.

Example

```
static U8 _aDataBuffer[512];

static void _Send(void) {
    unsigned NumBytes2Send;
    unsigned NumBytesRead;

    NumBytes2Send = _GetNumBytes2Send();
    while (NumBytes2Send >= sizeof(_aDataBuffer)) {
        NumBytesRead = _GetData(&_aDataBuffer[0], sizeof(_aDataBuffer));
        USB_BULK_WriteEx(&_aDataBuffer[0], NumBytesRead, 0);
        NumBytes2Send -= NumBytesRead;
    }
    USB_BULK_WriteEx(&_aDataBuffer[0], NumBytes2Send, 1);
}
```

5.5.2.16 USB_BULK_WriteExTimed()

Description

Sends data to the host with the option to send a zero-length packet at the end of the data transfer and a time-out option.

Prototype

```
int USB_BULK_WriteEx(const void* pData,
                    unsigned   NumBytes,
                    char       Send0PacketIfRequired
                    unsigned   ms);
```

Parameter	Description
pData	Pointer to a buffer that contains the written data.
NumBytes	Number of bytes to write.
Send0PacketIfRequired	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 byte. For high-speed devices the normal packet size is between 64-512 bytes.
ms	Time-out

Table 5.13: USB_BULK_ReadOverlapped() parameter list

Return value

Number of bytes that have been written within the given time-out.

Additional information

This function blocks a task until all data have been written or a time-out occurs. This function also returns when target is disconnected from host or when a USB reset occurred.

5.5.2.17 USB_BULK_WriteOverlapped()

Description

Write data to the host asynchronously.

Prototype

```
int USB_BULK_WriteOverlapped(const void* pData, unsigned NumBytes);
```

Parameter	Description
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.

Table 5.14: USB_BULK_WriteOverlapped() parameter list

Return value

Number of bytes that have already been sent to the HOST.

Additional information

This function will not block the calling task. The write transfer will be initiated and the function returns immediately. In order to synchronize, `USB_BULK_WaitForTX()` needs to be called.

5.5.2.18 USB_BULK_WriteOverlappedEx()

Description

Write data to the host asynchronously.

Prototype

```
int USB_BULK_WriteOverlappedEx(const void* pData,
                               unsigned    NumBytes,
                               char       Send0PacketIfRequired);
```

Parameter	Description
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.
Send0PacketIfRequired	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 byte. For high-speed devices the normal packet size is between 64-512 bytes.

Table 5.15: USB_BULK_WriteOverlappedEx() parameter list

Return value

Number of bytes that have already been sent to the HOST.

Additional information

This function will not block the calling task. The write transfer will be initiated and the function returns immediately. In order to synchronize, `USB_BULK_WaitForTX()` needs to be called.

5.5.2.19 USB_BULK_WriteTimed()

Description

Writes data from the host with a given time-out.

Prototype

```
int USB_BULK_WriteOverlapped(const void* pData,
                             unsigned   NumBytes,
                             unsigned   ms);
```

Parameter	Description
pData	Pointer to a buffer that contains the written data.
NumBytes	Number of bytes to write.
ms	Time-out

Table 5.16: USB_BULK_ReadOverlapped() parameter list

Return value

Number of bytes that have been written within the given time-out.

Additional information

This function blocks a task until all data have been written or a time-out occurs. This function also returns when target is disconnected from host or when a USB reset occurred.

5.5.2.20 USB_BULK_WriteNULLPacket()

Description

Sends a zero-length packet to the host.

Prototype

```
void USB_BULK_WriteNULLPacket(void);
```

Additional information

This function is useful to indicate that either no data are available or to indicate that this is the last packet of the data stream.

In normal cases sending a zero-length packets as a termination packet is not necessary since the stack handles this automatically when calling any USB_BULK write function (except for USB_BULK_WriteEx routines).

5.5.3 Data structures

5.5.3.1 USB_BULK_INIT_DATA

Description

Initialization structure which is required when adding a bulk interface to emUSB-Bulk.

Prototype

```
typedef struct {
    U8 EPIn;
    U8 EPOut;
} USB_BULK_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host

Table 5.17: USB_BULK_INIT_DATA elements

Example

Example excerpt from BULK_Echo1.c:

```
static void _AddBULK(void) {
    static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
    USB_BULK_INIT_DATA Init;

    Init.EPIn = USB_AddEP(1, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE, NULL, 0);
    Init.EPOut = USB_AddEP(0, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
        _abOutBuffer, USB_MAX_PACKET_SIZE);
    USB_BULK_Add(&Init);
}
```

5.5.3.2 USB_ON_RX_FUNC

Description

Callback function prototype that is used when calling the function

Prototype

```
typedef void USB_ON_RX_FUNC(const U8 * pData, unsigned NumBytes);
```

Member	Description
pData	Pointer to the data that have been received.
NumBytes	Number of bytes that have been received.

Table 5.18: USB_ON_RX_FUNC elements

5.6 Host API

This chapter describes the functions that can be used with the Windows host system.

General information

To communicate with the target USB-Bulk stack, the sample application project includes USB-Bulk specific source and header files (`USBBulk.c`, `USBULK.h`). These files contain API functions to communicate with the USB-Bulk target through the USB-Bulk driver.

Purpose of the USB Host API functions

To have an easy start-up when writing an application on the host side, these API functions have a simple interface and handle all required operations to communicate with the target USB-Bulk stack.

Therefore, all operations that need to open a channel, writing to or reading from the USB-Bulk stack are handled internally by the provided API functions.

Additional information can also be retrieved from the USB driver.

5.6.1 Host API list

The functions below are available on the host (Windows PC) side.

Function	Description
USB-Bulk basic functions	
<code>USBBULK_Open()</code>	Opens pipes to communicate with the first USB-Bulk device.
<code>USBBULK_OpenEx()</code>	Opens pipes to communicate with a specified USB-Bulk device.
<code>USBBULK_Close()</code>	Closes the pipes which are used for the communication with the first USB-Bulk device.
<code>USBBULK_CloseEx()</code>	Closes the pipes which are used for the communication to a specified USB-Bulk device.
USB-Bulk direct input/output functions	
<code>USBBULK_Read()</code>	Reads data from the first USB-Bulk device.
<code>USBBULK_ReadEx()</code>	Reads data from a specified USB-Bulk device.
<code>USBBULK_Write()</code>	Writes data to the first USB-Bulk device.
<code>USBBULK_WriteEx()</code>	Writes data to a specified USB-Bulk device.
<code>USBBULK_WriteRead()</code>	Reads and writes data from/to the first USB-Bulk device.
<code>USBBULK_WriteReadEx()</code>	Reads and writes data from/to a specified USB-Bulk device.
USB-Bulk control functions	
<code>USBBULK_GetDriverCompileDate()</code>	Gets the compile date and time of the USB-Bulk driver.
<code>USBBULK_GetDriverVersion()</code>	Retrieves the version of the USB-Bulk driver.
<code>USBBULK_GetConfigDescriptor()</code>	Gets the received target USB configuration descriptor of the first USB-Bulk device.
<code>USBBULK_GetConfigDescriptorEx()</code>	Gets the received target USB configuration descriptor of a specified USB-Bulk device.
<code>USBBULK_GetMode()</code>	Returns the read operation mode of the USB-Bulk device.
<code>USBBULK_GetModeEx()</code>	Returns the read operation mode of the USB-Bulk driver.
<code>USBBULK_GetNumAvailableDevices()</code>	Returns the number of connected USB-Bulk devices.
<code>USBBULK_GetReadMaxTransferSize()</code>	Retrieves the maximum transfer size of a read transaction the driver can receive from an application.
<code>USBBULK_GetReadMaxTransferSizeEx()</code>	Retrieves the maximum transfer size of a read transaction the driver can receive from an application.
<code>USBBULK_GetSN()</code>	Returns the serial number of the USB target device.
<code>USBBULK_GetWriteMaxTransferSize()</code>	Retrieves the maximum transfer size of a write transaction the driver can handle from an application.

Table 5.19: Host API function list

Function	Description
<code>USBBULK_GetWriteMaxTransferSizeEx()</code>	Retrieves the maximum transfer size of a write transaction the driver can handle from an application.
<code>USBBULK_SetMode()</code>	Sets the read operation mode of the USB-Bulk driver.
<code>USBBULK_SetModeEx()</code>	Sets the read operation mode of the USB-Bulk driver.
<code>USBBULK_SetTimeout()</code>	Sets a read time-out for a read transaction.
<code>USBBULK_SetTimeoutEx()</code>	Sets a read time-out for a read transaction.
<code>USBBULK_SetUSBId()</code>	Sets the vendor Id and product id that are used for connecting to the device.

Table 5.19: Host API function list

5.6.2 USB-Bulk Basic functions

5.6.2.1 USBULK_Open()

Description

Opens a read and write connection to the first connected target device using emUSB-Bulk.

Prototype

```
void * USBULK_Open(void);
```

Return value

'!= NULL' if a connection to the target running emUSB-Bulk could be established.
'== NULL' if a connection could not be established.

5.6.2.2 USBBULK_OpenEx()

Description

Opens a read and write connection to a specified device using the emUSB-Bulk kernel-mode driver.

Prototype

```
void * USBBULK_OpenEx(unsigned Id);
```

Parameter	Description
Id	Id number of the device [0..31].

Table 5.20: USBBULK_OpenEx() parameter list

Return value

'!= NULL' if a connection to the target running emUSB-Bulk could be established.

'== NULL' if a connection could not be established.

5.6.2.3 USBULK_Close()

Description

Closes all connections to the first target device using emUSB-Bulk.

Prototype

```
void USBULK_Close(void);
```

5.6.2.4 USBBULK_CloseEx()

Description

Closes all connections to a specified device using emUSB-Bulk.

Prototype

```
void USBBULK_CloseEx(unsigned Id);
```

Parameter	Description
Id	Id number of the device [0..31].

Table 5.21: USBBULK_CloseEx() parameter list

5.6.3 USB-Bulk direct input/output functions

5.6.3.1 USBULK_Read()

Description

Reads data from the first target device running emUSB-Bulk.

Prototype

```
int USBULK_Read(void * pBuffer, unsigned NumBytes);
```

Parameter	Description
<code>pBuffer</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.

Table 5.22: USBULK_Read() parameter list

Return value

'== `NumBytes`': All bytes have successfully been read.

'< `NumBytes`': A time-out occurred during read, when the emUSB driver is in normal mode, otherwise the emUSB driver returns the number of bytes that have been read from device.

'== -1': Cannot read from the device.

Additional information

`USBULK_Read()` sends the read request to the USB-Bulk driver. Because the driver can only read a certain amount of bytes from the device - the default value is 64 Kbytes; the driver will abort the transaction.

Therefore if `NumBytes` exceeds this limit, `USBULK_Read()` will read the desired `NumBytes` in chunks of the maximum read size the driver can handle.

5.6.3.2 USBULK_ReadEx()

Description

Reads data from a specified target device running emUSB-Bulk.

Prototype

```
int USBULK_ReadEx(unsigned Id, void * pBuffer, unsigned NumBytes);
```

Parameter	Description
Id	Id number of the device [0..31].
pBuffer	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 5.23: USBULK_ReadEx() parameter list

Return value

'== NumBytes': All bytes have successfully been read.

'< NumBytes': A time-out occurred during read, when the emUSB driver is in normal mode. Otherwise the emUSB driver returns the number of bytes that have been read from the device.

'== -1': Cannot read from device.

Additional information

USBULK_ReadEx() sends the read request to the emUSB driver. Because the driver can only read a certain amount of bytes from the device - the default value is 64 Kbytes; the driver will abort the transaction.

therefore, if NumBytes exceeds this limit, USBULK_Read() will read the desired Num-Bytes in chunks of the maximum read size the driver can handle.

5.6.3.3 USBULK_Write()

Description

Writes data to the first target device running emUSB-Bulk.

Prototype

```
int USBULK_Write(const void * pBuffer, unsigned NumBytes);
```

Parameter	Description
<code>pBuffer</code>	Pointer to a buffer to transfer.
<code>NumBytes</code>	Number of bytes to write.

Table 5.24: USBULK_Write() parameter list

Return value

'== `NumBytes`': All bytes have successfully been written.

NumBytes': A write error occurred.

Additional information

`USBULK_Write()` sends the write request to the emUSB driver. Because the driver can only write a certain amount of bytes to device - the default value is 64 Kbytes; the driver will abort the transaction.

Therefore if `NumBytes` exceeds this limit, `USBULK_Write()` will write the desired `NumBytes` in chunks of the maximum write size the driver can handle.

5.6.3.4 USBULK_WriteEx()

Description

Writes data to a specified target device running emUSB-Bulk.

Prototype

```
int USBULK_WriteEx(unsigned Id, const void * pBuffer, unsigned NumBytes);
```

Parameter	Description
Id	Id number of device [0..31].
pBuffer	Pointer to a buffer to transfer.
NumBytes	Number of bytes to write.

Table 5.25: USBULK_WriteEx() parameter list

Return value

'== NumBytes': All bytes have successfully been written.

Additional information

USBULK_WriteEx() sends the write request to the emUSB driver. Since the driver can only write a certain amount of bytes to the device - the default value is 64 Kbytes; the driver will abort the transaction.

Therefore if NumBytes exceeds this limit, USBULK_Write() will write the desired NumBytes in chunks of the maximum write size the driver can handle.

5.6.3.5 USBULK_WriteRead()

Description

Writes and reads data to and from the first target device running emUSB-Bulk.

Prototype

```
int USBULK_WriteRead(const void * pWrBuffer, unsigned WrNumBytes
                    void * pRdBuffer, unsigned RdNumBytes);
```

Parameter	Description
pWrBuffer	Pointer to a buffer to transfer.
WrNumBytes	Number of bytes to write.
pRdBuffer	Pointer to a buffer where the received data will be stored.
RdNumBytes	Number of bytes to read.

Table 5.26: USBULK_WriteRead() parameter list

Return value

'== NumBytes': All bytes have successfully been read after writing the data.

'< NumBytes': A time-out occurred during read, when the emUSB driver is in normal mode. Otherwise the emUSB driver returns the number of bytes that have been read from the device.

'== -1': Cannot read from the device after write.

Additional information

This function cannot be used with short mode enabled.

5.6.3.6 USBULK_WriteReadEx()

Description

Writes and reads data to and from specified target device running emUSB-Bulk.

Prototype

```
int USBULK_WriteReadEx(unsigned Id,
                       const void * pWrBuffer,
                       unsigned WrNumBytes,
                       void * pRdBuffer,
                       unsigned RdNumBytes);
```

Parameter	Description
Id	Id number of device [0..31].
pWrBuffer	Pointer to a buffer to transfer.
WrNumBytes	Number of bytes to write.
pRdBuffer	Pointer to a buffer where the received data will be stored.
RdNumBytes	Number of bytes to read.

Table 5.27: USBULK_WriteReadEx() parameter list

Return value

'== NumBytes': All bytes have successfully been read after writing the data.

'< NumBytes': A time-out occurred during read, when the emUSB driver is in normal mode, otherwise the emUSB driver returns the number of bytes that have been read from device.

'== -1' - Cannot read from the device after write.

Additional information

This function cannot be used with short mode enabled.

5.6.4 USB-Bulk Control functions

5.6.4.1 USBULK_GetDriverCompileDate()

Description

Gets the compile date and time of the emUSB bulk communication driver.

Prototype

```
unsigned USBULK_GetDriverCompileDate(char * s, unsigned Size);
```

Parameter	Description
s	Pointer to a buffer to store the compile date string.
Size	Size, in bytes, of the buffer pointed to by s.

Table 5.28: USBULK_GetDriverCompileDate() parameter list

Return value

If the function succeeds, the return value is nonzero and the buffer pointed by [s](#) contains the compile date and time of the emUSB driver in the standard format:

mm dd yyyy hh:mm:ss

If the function fails, the return value is zero.

5.6.4.2 USBBULK_GetDriverVersion()

Description

Retrieves the version of the emUSB bulk communication driver.

Prototype

```
unsigned USBBULK_GetDriverVersion(void);
```

Return value

If the function succeeds, the return value is the driver version of the driver as decimal value:

<Major Version><Minor Version><Subversion>. 24201 means 2.42a

If the function fails, the return value is zero; the version could not be retrieved.

5.6.4.3 USBULK_GetConfigDescriptor()

Description

Gets the received target USB configuration descriptor of the first device running emUSB-Bulk.

Prototype

```
int USBULK_GetConfigDescriptor(void * pBuffer, int Size);
```

Parameter	Description
pBuffer	Pointer to a buffer to store the config descriptor.
Size	Number of bytes to read.

Table 5.29: USBULK_GetConfigDescriptor() parameter list

Return value

If the function succeeds, the return value is nonzero and the buffer pointed by [pBuffer](#) contains the USB target device configuration descriptor.

If the function fails, the return value is zero.

5.6.4.4 USBULK_GetConfigDescriptorEx()

Description

Gets the received target USB configuration descriptor of a specified device running emUSB-Bulk.

Prototype

```
int USBULK_GetConfigDescriptor(unsigned Id, void * pBuffer, int Size);
```

Parameter	Description
Id	Id number of the device [0..31].
pBuffer	Pointer to a buffer to store the config descriptor.
Size	Number of bytes to read.

Table 5.30: USBULK_GetConfigDescriptorEx() parameter list

Return value

If the function succeeds, the return value is nonzero and the buffer pointed by `pBuffer` contains the USB target device configuration descriptor.

If the function fails, the return value is zero.

5.6.4.5 USBULK_GetMode()

Description

Returns the read operation mode of the driver for the first device running emUSB-Bulk.

Prototype

```
unsigned USBULK_GetMode(void);
```

Return value

USBULK_MODE_BIT_ALLOW_SHORT_READ - Short read mode is enabled.
0 - Short read mode is disabled.

5.6.4.6 USBULK_GetModeEx()

Description

Returns the read operation mode of the driver for a specified device running emUSB-Bulk.

Prototype

```
unsigned USBULK_GetModeEx(unsigned Id);
```

Parameter	Description
Id	Id number of device [0..31].

Table 5.31: USBULK_GetModeEx() parameter list

Return value

USBULK_MODE_BIT_ALLOW_SHORT_READ - Short read mode is enabled.
0 - Short read mode is disabled.

5.6.4.7 USBULK_GetNumAvailableDevices()

Description

Returns the number of connected USB-Bulk devices.

Prototype

```
unsigned USBULK_GetNumAvailableDevices(U32 * pMask);
```

Parameter	Description
<code>pMask</code>	Pointer to a U32 variable to receive the connected device mask. This parameter can be <code>NULL</code> .

Table 5.32: USBULK_GetNumAvailableDevices() parameter list

Return value

If the function succeeds, the return value is the number of available devices running emUSB-Bulk. For each emUSB device that is connected, a bit in `pMask` is set. For example if device 0 and device 2 are connected to the host, the value `pMask` points to will be `0x00000005`.

If the function fails, the return value is zero.

5.6.4.8 USBULK_GetReadMaxTransferSize()

Description

Retrieves the maximum transfer size of a read transaction the driver can receive from an application for the first device running emUSB-Bulk.

Prototype

```
unsigned USBULK_GetReadMaxTransferSize(void);
```

Return value

If the function succeeds, the return value is the maximum transfer size in bytes the driver can accept from an application.

If the function fails, the return value is zero.

5.6.4.9 USBULK_GetReadMaxTransferSizeEx()

Description

Retrieves the maximum transfer size of a read transaction the driver can receive from an application for a specified device running emUSB-Bulk.

Prototype

```
unsigned USBULK_GetReadMaxTransferSizeEx(unsigned Id);
```

Parameter	Description
Id	Id number of device [0..31].

Table 5.33: USBULK_GetReadMaxTransferSizeEx() parameter list

Return value

If the function succeeds, the return value is the maximum transfer size in bytes the driver can accept from an application.

If the function fails, the return value is zero.

5.6.4.10 USBBULK_GetSN()

Description

Retrieves the USB serial number as a string that was returned by the device during the enumeration.

Prototype

```
int USBBULK_GetSN(unsigned Id, char * pBuffer, unsigned NumBytes);
```

Parameter	Description
Id	Id number of device [0..31].
pBuffer	Pointer to a buffer to store the serial number of the device.
NumBytes	Size of the buffer in bytes.

Table 5.34: USBBULK_GetSN() parameter list

Return value

If the function succeeds, the return value is nonzero and the buffer pointed by `pBuffer` contains the serial number of the device running emUSB-Bulk.

If the function fails, the return value is zero.

5.6.4.11 USBULK_GetWriteMaxTransferSize()

Description

Retrieves the maximum transfer size of a write transaction the driver can handle from an application (for the first device running emUSB-Bulk).

Prototype

```
unsigned USBULK_GetWriteMaxTransferSize(void);
```

Return value

If the function succeeds, the return value is the maximum transfer size in bytes the driver can accept from an application to send data to the target device.
If the function fails, the return value is zero.

5.6.4.12 USBULK_GetWriteMaxTransferSizeEx()

Description

Retrieves the maximum transfer size of a write transaction the driver can handle from an application for a specified device running emUSB-Bulk.

Prototype

```
unsigned USBULK_GetWriteMaxTransferSizeEx(unsigned Id);
```

Parameter	Description
Id	Id number of device [0..31].

Table 5.35: USBULK_GetWriteMaxtransferSizeEx() parameter list

Return value

If the function succeeds, the return value is the maximum transfer size in bytes the driver can accept from an application to send data to the target device.

If the function fails, the return value is zero.

5.6.4.13 USBULK_SetMode()

Description

Sets the read operation mode of the driver for a device running emUSB-Bulk.

Prototype

```
unsigned USBULK_SetMode(unsigned Mode);
```

Parameter	Description
Mode	Read and write mode for the USB-Bulk driver. This is a combination of the following flags, combined by binary OR: USBULK_MODE_BIT_ALLOW_SHORT_READ

Table 5.36: USBULK_SetMode() parameter list

Return value

If the function succeeds, the return value is nonzero. The read and write mode for the driver has been successfully set.

If the function fails, the return value is zero.

Additional information

USBULK_MODE_BIT_ALLOW_SHORT_READ allows short read transfers. Short transfers are transfers of less bytes than requested. If this bit is specified, the read function USBULK_Read() returns as soon as data is available, even if it is just a single byte.

Example

```
static void _TestMode(void) {
    unsigned Mode;
    char * pText;

    Mode = USBULK_GetMode();
    if (Mode & USBULK_MODE_BIT_ALLOW_SHORT_READ) {
        pText = "USE_SHORT_MODE";
    } else {
        pText = "USE_NORMAL_MODE";
    }
    printf("USB-Bulk driver is in %s\n", pText);
    printf("Set mode to USBULK_MODE_BIT_ALLOW_SHORT_READ\n");
    USBULK_SetMode(USBULK_MODE_BIT_ALLOW_SHORT_READ);
    Mode = USBULK_GetMode();
    if (Mode & USBULK_MODE_BIT_ALLOW_SHORT_READ) {
        pText = "USE_SHORT_MODE";
    } else {

        pText = "USE_NORMAL_MODE";
    }
    printf("USB-Bulk driver is now in %s\n", pText);
}
```

5.6.4.14 USBBULK_SetModeEx()

Description

Sets the read operation mode of the driver for a specified device running emUSB-Bulk.

Prototype

```
unsigned USBBULK_SetModeEx(unsigned Id, unsigned Mode);
```

Parameter	Description
Id	Id of the device.
Mode	Read and write mode for the USB-Bulk driver. This is a combination of the following flags, combined by binary or: USBBULK_MODE_BIT_ALLOW_SHORT_READ

Table 5.37: USBBULK_SetModeEx() parameter list

Return value

If the function succeeds, the return value is nonzero. The read and write mode for the driver has been successfully set.

If the function fails, the return value is zero.

Additional information

USBBULK_MODE_BIT_ALLOW_SHORT_READ allows short read transfers. Short transfers are transfers of less bytes than requested. If this bit is specified, the read function USBBULK_ReadEx() returns as soon as data is available, even if it is just a single byte.

Example

```
static void _TestModeEx(unsigned DeviceId) {
    unsigned Mode;
    char * pText;

    Mode = USBBULK_GetModeEx(DeviceId);
    if (Mode & USBBULK_MODE_BIT_ALLOW_SHORT_READ) {
        pText = "USE_SHORT_MODE";
    } else {
        pText = "USE_NORMAL_MODE";
    }
    printf("USB-Bulk driver is in %s for device %d\n", pText, DeviceId);
    printf("Set mode to USBBULK_MODE_BIT_ALLOW_SHORT_READ\n");
    USBBULK_SetModeEx(DeviceId, USBBULK_MODE_BIT_ALLOW_SHORT_READ);
    Mode = USBBULK_GetModeEx(DeviceId);
    if (Mode & USBBULK_MODE_BIT_ALLOW_SHORT_READ) {
        pText = "USE_SHORT_MODE";
    } else {
        pText = "USE_NORMAL_MODE";
    }
    printf("USB-Bulk driver is now in %s for device %d\n", pText, DeviceId);
}
```

5.6.4.15 USBBULK_SetTimeout()

Description

Sets a read time-out for a read operation to the first device running emUSB-Bulk.

Prototype

```
void USBBULK_SetTimeout(int Timeout);
```

Parameter	Description
Timeout	Timeout in milliseconds set for a read operation.

Table 5.38: USBBULK_SetTimeout() parameter list

5.6.4.16 USBULK_SetTimeoutEx()

Description

Sets a read time-out for a read operation.

Prototype

```
void USBULK_SetTimeout(unsigned Id, int Timeout);
```

Parameter	Description
Id	Id number of device [0..31].
Timeout	Timeout in milliseconds set for a read operation.

Table 5.39: USBULK_SetTimeOutEx() parameter list

5.6.4.17 USBBULK_SetUSBId()

Description

Sets the Vendor id and product id that are used for connecting to the device.

Prototype

```
void USBBULK_SetUSBId(U16 VendorId, U16 ProductId);
```

Parameter	Description
VendorId	The vendor id that was assigned by USB.org.
ProductId	The product id that is used for the device.

Table 5.40: USBBULK_SetUSBId() parameter list

Additional information

It is necessary to call this function first before opening any connection to the device. The initial values for these IDs are:

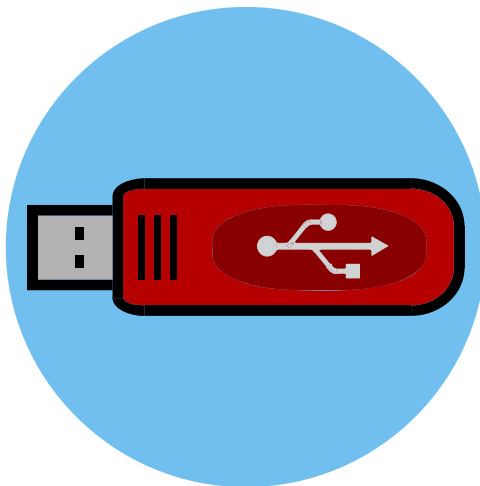
VendorId = 0x8765

ProductId = 0x1234

Chapter 6

Mass Storage Device Class (MSD)

This chapter gives a general overview of the MSD class and describes how to get the MSD component running on the target.



6.1 Overview

The Mass Storage Device (MSD) is a USB class protocol defined by USB Implementers Forum. The class itself is used to get access to one storage medium or multiple storage mediums.

As the USB mass storage device class is well standardized, every major OS such as Microsoft Windows operating systems (Windows ME, Windows 2000, Windows XP, Windows 2003 and Windows Vista), Apple Mac OS X, Linux and many more supports it. So therefore an installation of a custom-host USB driver is normally not necessary.

emUSB-MSD comes as a whole packet and contains the following:

- Generic USB handling
- An optional target USB driver
- MSD device class implementation, including support for direct disk and CD-ROM mode (CD-ROM access is separate component)
- Several storage drivers for handling different devices
- Example applications with different configuration storage driver

6.2 Configuration

6.2.1 Initial configuration

To get emUSB-MSD up and running as well as doing an initial test, the configuration as it is delivered should not be modified.

6.2.2 Final configuration

The configuration must only be modified, when emUSB should be used in your final product. Refer to section *Configuration* on page 41 to get detailed information about the generic information functions which has to be adapted.

In order to comply with Mass Storage Device Bootability spec, the function `USB_GetSerialNumber()` should return a string with at least 12 characters, where as each character should represent a hexadecimal character.

6.2.3 Class specific configuration functions

Beside the generic emUSB-MSD configuration functions, three additional functions can be adapted before the emUSB MSD component should be used in a final product. Example implementations of these functions are supplied in the MSD example application `MSD_FS_Start.c`, located in the `Application` directory of emUSB.

Function	Description
emUSB-MSD configuration functions	
<code>USB_MSD_GetVendorName()</code>	Returns the manufacturer name.
<code>USB_MSD_GetProductName()</code>	Returns the MSD product name.
<code>USB_MSD_GetProductVer()</code>	Returns the product version of the MSD device.
<code>USB_MSD_GetSerialNo()</code>	Returns the serial number of the MSD device.

Table 6.1: List of class specific configuration functions

6.2.3.1 USB_MSD_GetVendorName()

Description

Should return the vendor name of the mass storage device.

Prototype

```
const char * USB_MSD_GetVendorName(U8 Lun);
```

Parameter	Description
Lun	Specifies the logical unit number whose vendor name should be returned.

Table 6.2: USB_MSD_GetVendorName() parameter list

Example

```
const char * USB_MSD_GetVendorName(U8 Lun) {
    return "Vendor";
}
```

Additional information

The manufacturer name is used during the enumeration phase. Together with the product name and the serial number should it give detailed information to the user about which device is connected to the device. The string should be no longer than 8 bytes.

6.2.3.2 USB_MSD_GetProductName()

Description

Should return the product name of the mass storage device.

Prototype

```
const char * USB_MSD_GetProductName(U8 Lun);
```

Parameter	Description
Lun	Specifies the logical unit number those product name should be returned.

Table 6.3: USB_MSD_GetProductName() parameter list

Example

```
const char * USB_GetProductName(U8 Lun) {
    return "MSD device";
}
```

Additional information

The product name string should be not longer than 16 byte.

6.2.3.3 USB_MSD_GetProductVer()

Description

Should return the product version number of the mass storage device.

Prototype

```
const char * USB_MSD_GetProductVer(U8 Lun);
```

Parameter	Description
Lun	Specifies the logical unit number those version should be returned.

Table 6.4: USB_MSD_GetProductVer() parameter list

Example

```
const char * USB_MSD_GetProductVer(U8 Lun) {  
    return "1.00";  
}
```

Additional information

The product version string should be no longer than 8 bytes.

6.2.3.4 USB_MSD_GetSerialNo()

Description

Should return the product serial number of the mass storage device.

Prototype

```
const char * USB_MSD_GetSerialNo(U8 Lun);
```

Parameter	Description
Lun	Specifies the logical unit number those serial number should be returned.

Table 6.5: USB_MSD_GetSerialNo() parameter list

Example

```
const char * USB_MSD_GetSerialNo(U8 Lun) {
    return "1234657890";
}
```

Additional information

The product version string should be no longer than 10 bytes.

6.2.4 Running the example application

The directory `Application` contains example applications that can be used with `emUSB` and the `MSD` component. To test the `emUSB-MSD` component, build and download the application of choice into the target. Remove the USB connection and reconnect the target to the host. The target will enumerate and can be accessed via a file browser.

6.2.4.1 MSD_Start_StorageRAM.c in detail

The main part of the example application `USB_MSD_Start_StorageRAM.c` is implemented in a single task called `MainTask()`.

```
/* MainTask() - excerpt from USB_MSD_Start_StorageRAM.c */
void MainTask(void);
void MainTask(void) {
    USB_Init();
    _AddMSD();
    USB_Start();
    while (1) {
        while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        BSP_SetLED(0);
        USB_MSD_Task();
    }
}
```

The first step is to initialize the USB core stack using `USB_Init()`. The function `_AddMSD()` configures all required endpoints and assigns the used storage medium to the `MSD` component.

```
/* _AddMSD() - excerpt from MSD_Start_StorageRAM.c */
static void _AddMSD(void) {
    static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
    USB_MSD_INIT_DATA    InitData;
    USB_MSD_INST_DATA    InstData;

    InitData.EPIn  = USB_AddEP(1, USB_TRANSFER_TYPE_BULK,
                               USB_MAX_PACKET_SIZE, NULL, 0);
    InitData.EPOut = USB_AddEP(0, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
                               _abOutBuffer, USB_MAX_PACKET_SIZE);
    USB_MSD_Add(&InitData);
    //
    // Add logical unit 0: RAM drive
    //
    memset(&InstData, 0, sizeof(InstData));
    InstData.pAPI          = &USB_MSD_StorageRAM;
    InstData.DriverData.pStart = (void*)MSD_RAM_ADDR;
    InstData.DriverData.NumSectors = MSD_RAM_NUM_SECTORS;
    InstData.DriverData.SectorSize = MSD_RAM_SECTOR_SIZE;
    USB_MSD_AddUnit(&InstData);
}
```

The example application uses a RAM disk as storage medium.

The example RAM disk has a size of 23 KBytes (46 sectors with a sector size of 512 bytes). You can increase the size of the RAM disk by modifying the macros `MSD_RAM_NUM_SECTORS` and `MSD_RAM_SECTOR_SIZE`, but the size must be at least 23 KBytes otherwise a Windows host cannot format the disk.

```
/* AddMSD() - excerpt from MSD_Start_StorageRAM.c */
#define MSD_RAM_NUM_SECTORS    46
#define MSD_RAM_SECTOR_SIZE    512
```

6.3 Target API

Function	Description
API functions	
<code>USB_MSD_Add()</code>	Adds an MSD-class interface to the USB stack.
<code>USB_MSD_AddUnit()</code>	Adds a mass storage device to the emUSB-MSD.
<code>USB_MSD_AddCDRom()</code>	Adds a CD-ROM device to the emUSB-MSD.
<code>USB_MSD_SetPreventAllowRemovalHook()</code>	Sets a callback function to prevent/allow removal of storage medium.
<code>USB_MSD_SetReadWriteHook()</code>	Sets a callback function which is called with every read or write access to the storage medium.
<code>USB_MSD_Task()</code>	Handles the MSD-specific protocol.
Extended API functions	
<code>USB_MSD_Connect()</code>	Connects the storage medium to the MSD.
<code>USB_MSD_Disconnect()</code>	Disconnects the storage medium from the MSD.
<code>USB_MSD_RequestDisconnect()</code>	Sets the DisconnectRequest flag.
<code>USB_MSD_UpdateWriteProtect()</code>	Updates the IsWriteProtected flag for a storage medium.
<code>USB_MSD_WaitForDisconnection()</code>	Waits for disconnection while time out is not reached.
Data structures	
<code>USB_MSD_INIT_DATA</code>	emUSB-MSD initialization structure that is needed when adding an MSD interface.
<code>USB_MSD_INFO</code>	emUSB-MSD storage information.
<code>USB_MSD_INST_DATA</code>	Structure that is used when adding a device to emUSB-MSD.
<code>PREVENT_ALLOW_REMOVAL_HOOK</code>	Callback invoked when the storage medium is removed.
<code>READ_WRITE_HOOK</code>	Callback invoked when accessing the storage medium.
<code>USB_MSD_INST_DATA_DRIVER</code>	Structure that is passed to the driver.
<code>USB_MSD_STORAGE_API</code>	Structure that contains callbacks to the storage driver.

Table 6.6: List of emUSB MSD interface functions and data structures

6.3.1 API functions

6.3.1.1 USB_MSD_Add()

Description

Adds an MSD-class interface to the USB stack.

Prototype

```
void USB_MSD_Add      (const USB_MSD_INIT_DATA * pInitData);
```

Parameter	Description
pInitData	Pointer to a <code>USB_MSD_INIT_DATA</code> structure.

Table 6.7: USB_MSD_Add() parameter list

Additional information

After the initialization of general emUSB, this is the first function that needs to be called when an MSD interface is used with emUSB. The structure `USB_MSD_INIT_DATA` has to be initialized before `USB_MSD_Add()` is called. Refer to `USB_MSD_INIT_DATA` on page 159 for more information.

6.3.1.2 USB_MSD_AddUnit()

Description

Adds a mass storage device to emUSB-MSD.

Prototype

```
void USB_MSD_AddUnit (const USB_MSD_INST_DATA * pInstData);
```

Parameter	Description
<code>pInstData</code>	Pointer to a <code>USB_MSD_INST_DATA</code> structure that is used to add the desired drive to the USB-MSD stack.

Table 6.8: USB_MSD_AddUnit() parameter list

Additional information

It is necessary to call this function right after `USB_MSD_Add()` was called.

This function will then add a R/W storage device such as a hard drive, MMC/SD cards or NAND flash etc., to emUSB-MSD, which then will be used to exchange data with the host. The structure `USB_MSD_INST_DATA` has to be initialized before `USB_MSD_AddUnit()` is called. Refer to `USB_MSD_INST_DATA` on page 161 for more information.

6.3.1.3 USB_MSD_AddCDRom()

Description

Adds a CD-ROM device to emUSB-MSD.

Prototype

```
void USB_MSD_AddCDRom(const USB_MSD_INST_DATA * pInstData);
```

Parameter	Description
<code>pInstData</code>	Pointer to a <code>USB_MSD_INST_DATA</code> structure that is used to add the desired drive to the USB-MSD stack.

Table 6.9: USB_MSD_AddCDRom() parameter list

Additional information

Similar to `USB_MSD_AddUnit()`, this function should be called after `USB_MSD_Add()` was called. The structure `USB_MSD_INST_DATA` has to be initialized before `USB_MSD_AddUnit()` is called. Refer to `USB_MSD_INST_DATA` on page 161 for more information.

6.3.1.4 USB_MSD_SetPreventAllowRemovalHook()

Description

Sets a callback function to prevent/allow removal of storage medium.

Prototype

```
void USB_MSD_SetPreventAllowRemovalHook(U8 Lun,
                                         PREVENT_ALLOW_REMOVAL_HOOK * pfOnPreventAllowRemoval)
```

Parameter	Description
pfOnPreventAllowRemoval	Pointer to the callback function PREVENT_ALLOW_REMOVAL_HOOK. For detailed information about the function pointer, refer to <i>PREVENT_ALLOW_REMOVAL_HOOK</i> on page 162.

Table 6.10: USB_MSD_SetPreventAllowRemovalHook() parameter list

6.3.1.5 USB_MSD_SetReadWriteHook()

Description

Sets a callback function which gives information about the read and write operation to the storage medium.

Prototype

```
void USB_MSD_SetReadWriteHook(U8 Lun, READ_WRITE_HOOK * pfOnReadWrite)
```

Parameter	Description
<code>pfOnReadWrite</code>	Pointer to the callback function <code>READ_WRITE_HOOK</code> . For detailed information about the function pointer, refer to <code>READ_WRITE_HOOK</code> on page 163.

Table 6.11: USB_MSD_SetReadWriteHook() parameter list

6.3.1.6 USB_MSD_Task()

Description

Task that handles the MSD-specific protocol.

Prototype

```
void USB_MSD_Task(void);
```

Additional information

After the USB device has been successfully enumerated and configured, the `USB_MSD_Task()` should be called. When the device is detached or is suspended, `USB_MSD_Task()` will return.

6.3.2 Extended API functions

6.3.2.1 USB_MSD_Connect()

Description

Connects the storage medium to the MSD.

Prototype

```
void USB_MSD_Connect(U8 Lun);
```

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.

Table 6.12: USB_MSD_Connect() parameter list

Additional information

The storage medium is initially always connected to the MSD component. This function is normally used, when the the storage medium is disconnected in order to do some internal file system operation.

6.3.2.2 USB_MSD_Disconnect()

Description

Disconnects the storage medium from the MSD.

Prototype

```
void USB_MSD_Disconnect(U8 Lun);
```

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.

Table 6.13: USB_MSD_Disconnect() parameter list

Additional information

This function will force the storage medium to be disconnected. The host will be informed that the medium is not present. In order to reconnect back the device to the host, the function `USB_MSD_Connect()` should be used.

6.3.2.3 USB_MSD_RequestDisconnect()

Description

Sets the DisconnectRequest flag.

Prototype

```
void USB_MSD_RequestDisconnect(U8 Lun);
```

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.

Table 6.14: USB_MSD_RequestDisconnect() parameter list

Additional information

This function sets the a disconnect flag for the storage medium. As soon as the next MSD command is sent to the device, the host will be informed that the device is currently not available. To reconnect the storage medium, `USB_MSD_Connect()` should be called.

6.3.2.4 USB_MSD_UpdateWriteProtect()

Description

Updates the IsWriteProtected flag for a storage medium.

Prototype

```
void USB_MSD_UpdateWriteProtect(U8 Lun, U8 IsWriteProtected);
```

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.
IsWriteProtected	1 - Medium is write-protected. 0 - Medium is not write-protected.

Table 6.15: USB_MSD_UpdateWriteProtect() parameter list

Additional information

This functions allows to update the write-protect status of the storage-medium. Please make sure that this function is called when the LUN is disconnected from the HOST, otherwise the WriteProtected flag is normally not recognized.

6.3.2.5 USB_MSD_WaitForDisconnection()

Description

Waits for disconnection while time out is not reached.

Prototype

```
int USB_MSD_WaitForDisconnection(U8 Lun, U32 TimeOut);
```

Parameter	Description
Lun	0-based index for the unit number. Using only one storage medium, this parameter is 0.
TimeOut	Time-out given in ms (timer ticks).

Table 6.16: USB_MSD_WaitForDisconnection() parameter list

Return value

- 0 - Error: Time-out reached. Storage medium is not disconnected.
- 1 - Success: Storage medium is disconnected.

Additional information

The stack disconnects the storage medium next time when the HOST requests the status of the storage medium. Win2k does not periodically check the status of a USB MSD. Therefore, the time out is required to leave the loop. The return value can be used to decide if the disconnection should be forced. In this case, `USB_MSD_Disconnect()` should be called.

6.3.3 Data structures

6.3.3.1 USB_MSD_INIT_DATA

Description

emUSB-MSD initialization structure that is required when adding an MSD interface.

Prototype

```
typedef struct {
    U8 EPIn;
    U8 EPOut;
    U8 InterfaceNum;
} USB_MSD_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
InterfaceNum	Interface number. This member is normally internally used, so therefore the value should be set to 0.

Table 6.17: USB_MSD_INIT_DATA elements

Additional Information

This structure holds the endpoints that should be used with the MSD interface. Refer to *USB_AddEP()* on page 57 for more information about how to add an endpoint.

6.3.3.2 USB_MSD_INFO

Description

emUSB-MSD storage interface.

Prototype

```
typedef struct {  
    U32 NumSectors;  
    U16 SectorSize;  
} USB_MSD_INFO;
```

Member	Description
NumSectors	Number of available sectors.
SectorSize	Size of one sector.

Table 6.18: USB_MSD_INFO elements

6.3.3.3 USB_MSD_INST_DATA

Description

Structure that is used when adding a device to emUSB-MSD.

Prototype

```
typedef struct {
    const USB_MSD_STORAGE_API * pAPI;
    USB_MSD_INST_DATA_DRIVER    DriverData;
    U8                           DeviceType;
    U8                           IsPresent;
    USB_MSD_HANDLE_CMD          * pfHandleCmd;
    U8                           IsWriteProtected;
} USB_MSD_INST_DATA;
```

Member	Description
pAPI	Pointer to a structure that holds the storage device driver API.
DriverData	Driver data that are passed to the storage driver. Refer to <i>USB_MSD_INST_DATA_DRIVER</i> on page 164 for detailed information about how to initialize this structure.
DeviceType	Determines the type of the device.
IsPresent	Determines if the medium is storage is present. For non-removable devices always 1.
pfHandleCmd	Optional pointer to a callback function which handles SCSI commands. typedef U8 (USB_MSD_HANDLE_CMD) (U8 Lun);
IsWriteProtected	Specifies whether the storage medium should be write-protected.

Table 6.19: USB_MSD_INST_DATA elements

Additional Information

All non-optional members of this structure need to be initialized correctly, except `Device Type` because it is done by the functions `USB_MSD_AddUnit()` or `USB_MSD_AddCDROM()`.

6.3.3.4 PREVENT_ALLOW_REMOVAL_HOOK

Description

Callback function to prevent/allow removal of storage medium.

Prototype

```
typedef void (PREVENT_ALLOW_REMOVAL_HOOK) (U8 PreventRemoval);
```

6.3.3.5 READ_WRITE_HOOK

Description

Callback function which is called with every read/write access to the storage medium.

Prototype

```
typedef void (READ_WRITE_HOOK) (U8 Lun,  
                                U8 IsRead,  
                                U8 OnOff,  
                                U32 StartLBA,  
                                U32 NumBlocks);
```

6.3.3.6 USB_MSD_INST_DATA_DRIVER

Description

USB-MSD initialization structure that is required when adding an MSD interface.

Prototype

```
typedef struct {
    void      * pStart;
    U32       StartSector;
    U32       NumSectors;
    U32       SectorSize;
    void      * pSectorBuffer;
    unsigned   NumBytes4Buffer;
} USB_MSD_INST_DATA_DRIVER;
```

Member	Description
pStart	A pointer defining the start address.
StartSector	The start sector that is used for the driver.
NumSectors	The available number of sectors available for the driver.
SectorSize	The sector size that should be used by the driver.
pSectorBuffer	Pointer to a application provided buffer to be used as temporary buffer for storing the sector data
NumBytes4Buffer	Size of the application provided buffer.

Table 6.20: USB_MSD_INST_DATA_DRIVER

Additional Information

This structure is passed to the storage driver. Therefore, the member of this structure can depend on the driver that is used.

For the storage driver that are shipped with this software the member of USB_MSD_INST_DATA_DRIVER have the following

USB_MSD_StorageRAM:

Member	Description
pStart	A pointer defining the start address of the RAM disk.
StartSector	This member is ignored.
NumSectors	The available number of sectors available for the RAM disk.
SectorSize	The sector size that should be used by the driver.

USB_MSD_StorageByName:

Member	Description
pStart	Pointer to a string holding the name of the volumes that should be used, for example "nand:" "mmc:1:"
StartSector	Specifies the start sector.
NumSectors	Number of sectors that should be used.
SectorSize	This member is ignored.
pSectorBuffer	Pointer to a application provided buffer to be used as temporary buffer for storing the sector data
NumBytes4Buffer	Size of the application provided buffer. Please make sure that the buffer can at least 3 sectors otherwise, pSectorBuffer and NumBytes4Buffer are ignored and an internal sector buffer is used. This sector-buffer is then allocated by using the FS-Storage-Layer functions.

6.3.3.7 USB_MSD_STORAGE_API

Description

Structure that contains callbacks to the storage driver.

Prototype

```
typedef struct {
    void (*pfInit)                (U8          Lun,
                                   const USB_MSD_INST_DATA_DRIVER * pDriverData);

    void (*pfGetInfo)            (U8          Lun,
                                   USB_MSD_INFO * pInfo);

    U32 (*pfGetReadBuffer)      (U8          Lun,
                                   U32          SectorIndex,
                                   void         ** ppData,
                                   U32          NumSectors);

    char (*pfRead)               (U8          Lun,
                                   U32          SectorIndex,
                                   void         * pData,
                                   U32          NumSector);

    U32 (*pfGetWriteBuffer)     (U8          Lun,
                                   U32          SectorIndex,
                                   void         ** ppData,
                                   U32          NumSectors);

    char (*pfWrite)              (U8          Lun,
                                   U32          SectorIndex,
                                   const void *  pData,
                                   U32          NumSectors);

    char (*pfMediumIsPresent)   (U8          Lun);

    void (*pfDeInit)            (U8          Lun);
} USB_MSD_STORAGE_API;
```

Member	Description
pfInit	Initializes the storage medium.
pfGetInfo	Retrieves storage medium information such as sector size and number of sectors available.
pfGetReadBuffer	Prepares read function and returns a pointer to a buffer that is used by the storage driver.
pfRead	Reads one or multiple sectors from the storage medium.
pfGetWriteBuffer	Prepares write function and returns a pointer to a buffer that is used by the storage driver.
pfWrite	Writes one or more sectors to the storage medium.
pfMediumIsPresent	Checks if medium is present.
pfDeInit	Deinitializes the storage medium.

Table 6.21: List of callback functions of USB_MSD_STORAGE_API

Additional Information

USB_MSD_STORAGE_API is used to retrieve information from the storage device driver or access data that need to be read or written. Detailed information can be found in *Storage Driver* on page 166.

6.4 Storage Driver

This section describes the storage interface in detail.

6.4.1 General information

The storage interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization. Its implementation handles the details of how data is actually read from or written to memory.

Additionally, MSD knows two different media types:

- Direct media access, for example RAM-Disk, NAND flash, MMC/SD cards etc.
- CD-ROM emulation.

6.4.1.1 Supported storage types

The supported storage types include:

- RAM, directly connected to the processor via the address bus.
- External flash memory, e.g. SD cards.
- Mechanical drives, for example CD-ROM. This is essentially an ATA/SCSI to USB bridge.

6.4.1.2 Storage drivers supplied with this release

This release comes with the following drivers:

- `USB_MSD_StorageRAM`: A RAM driver which should work with almost any device.
- `USB_MSD_StorageByIndex`: A storage driver that uses the storage layer (logical block layer) of emFile to access the device.
- `USB_MSD_StorageByName`: A storage driver that uses the storage layer (logical block layer) of emFile to access the device.

6.4.2 Interface function list

As described above, access to a storage media is realized through an API-function table (`USB_MSD_STORAGE_API`). The storage functions are declared in `USB\MSD\USB_MSD.h`. The structure is described in section *Data structures* on page 159.

6.4.3 USB_MSD_STORAGE_API in detail

6.4.3.1 (*pfInit)()

Description

Initializes the storage medium.

Prototype

```
void (*pfInit)(U8 Lun, const USB_MSD_INST_DATA_DRIVER * pDriverData);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pDriverData	Pointer to a <code>USB_MSD_INST_DATA_DRIVER</code> structure that contains all information that are necessary for the driver initialisation. For detailed information about the <code>USB_MSD_INST_DATA_DRIVER</code> structure, refer to <code>USB_MSD_INST_DATA_DRIVER</code> on page 164.

Table 6.22: (*pfInit)() parameter list

6.4.3.2 (*pfGetInfo)()

Description

Retrieves storage medium information such as sector size and number of sectors available.

Prototype

```
void (*pfGetInfo)(U8 Lun, USB_MSD_INFO * pInfo);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pInfo	Pointer to a <code>USB_MSD_INFO</code> structure. For detailed information about the <code>USB_MSD_INFO</code> structure, refer to <code>USB_MSD_INFO</code> on page 160.

Table 6.23: (*pfGetInfo)() parameter list

6.4.3.3 (*pfGetReadBuffer)()

Description

Prepares the read function and returns a pointer to a buffer that is used by the storage driver.

Prototype

```
U32 (*pfGetReadBuffer)(U8 Lun, U32 SectorIndex,
                      void ** ppData, U32 NumSectors);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the read operation.
ppData	Pointer to a pointer to store the read buffer address of the driver.
NumSectors	Number of sectors to read.

Table 6.24: (*pfGetReadBuffer)() parameter list

Return value

Number of sectors that can be read at once by the driver.

6.4.3.4 (*pfRead)()

Description

Reads one or multiple sectors from the storage medium.

Prototype

```
char (*pfRead)(U8 Lun, U32 SectorIndex, void * pData, U32 NumSector);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector from where the read operation is started.
pData	Pointer to buffer to store the read data.
NumSectors	Number of sectors to read.

Table 6.25: (*pfRead)() parameter list

Return value

0: Success

6.4.3.5 (*pfGetWriteBuffer)()

Description

Prepares the write function and returns a pointer to a buffer that is used by the storage driver.

Prototype

```
U32 (*pfGetWriteBuffer)(U8 Lun, U32 SectorIndex,
void ** ppData, U32 NumSectors);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the write operation.
ppData	Pointer to a pointer to store the write buffer address of the driver.
NumSectors	Number of sectors to write.

Table 6.26: (*pfGetWriteBuffer)() parameter list

Return value

Number of sectors that can be written into the buffer.

6.4.3.6 (*pfWrite)()

Description

Writes one or more sectors to the storage medium.

Prototype

```
char (*pfWrite)(U8 Lun, U32 SectorIndex,
                const void * pData, U32 NumSectors);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the write operation.
pData	Pointer to data to be written to the storage medium.
NumSectors	Number of sectors to write.

Table 6.27: (*pfWrite)() parameter list

Return value

0: Success

Any other value means error.

6.4.3.7 (*pfMediumIsPresent)()

Description

Checks if medium is present.

Prototype

```
char (*pfMediumIsPresent) (U8 Lun);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.

Table 6.28: (*pfMediumIsPresent)() parameter list

Return value

1: Medium is present.
0: Medium is not present.

6.4.3.8 (*pfDeInit)()

Description

Deinitializes the storage medium.

Prototype

```
void (*pfDeInit) (U8 Lun);
```

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.

Table 6.29: (*pfDeInit)() parameter list

Chapter 7

Communication Device Class (CDC)

This chapter describes how to get emUSB up and running as a CDC device.

7.1 Overview

The Communication Device Class (CDC) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol covers the handling of the following communication flows:

- VirtualCOM/Serial interface
- Universal modem device
- ISDN communication
- Ethernet communication

This implementation of CDC currently supports the virtual COM/Serial interface, thus the USB device will behave like a serial interface.

Normally, a custom USB driver is not necessary, because a kernel mode driver for USB-CDC serial communication is delivered by major Microsoft Windows operating systems. For installing the USB-CDC serial device an `.inf` file is needed, which is also delivered. Linux handles USB 2 virtual COM ports since Kernel Ver. 2.4. Further information can be found in the Linux Kernel documentation.

7.1.1 Configuration

The configuration section will later on be modified to match the real application. For the purpose of getting emUSB up and running as well as doing an initial test, the configuration as delivered should not be modified.

7.2 The example application

The start application (in the `Application` subfolder) is a simple echo server, which can be used to test emUSB. The application receives data byte by byte and sends it back to the host.

Source code of `USB_CDC_Start.c`:

```

/*****
*           SEGGER Microcontroller GmbH & Co. KG
*   Solutions for real time microcontroller applications
*****/
File      : USB_CDC_Start.c
Purpose   : Start Application for using the device as CDC device
-----  END-OF-HEADER -----*/

#include <stdio.h>
#include "BSP.h"
#include "USB.h"
#include "USB_CDC.h"

/*****
*
*   Static code
*
*****/
*/

/*****
*
*   _OnLineCoding
*
*   Function description
*   Called whenever a "SetLineCoding" Packet has been received
*
*   Notes
*   (1) Context
*       This function is called directly from an ISR in most cases.
*/
static void _OnLineCoding(USB_CDC_LINE_CODING * pLineCoding) {
#if 0
    printf("DTERate=%u, CharFormat=%u, ParityType=%u, DataBits=%u\n",
           pLineCoding->DTERate,
           pLineCoding->CharFormat,
           pLineCoding->ParityType,
           pLineCoding->DataBits);
#endif
}

/*****
*
*   _AddCDC
*
*   Function description
*   Add communication device class to USB stack
*/
static void _AddCDC(void) {
    static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
    USB_CDC_INIT_DATA    InitData;

    InitData.EPOut = USB_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_BULK, 0,
                               _abOutBuffer, USB_MAX_PACKET_SIZE);
}

```

```

InitData.EPIn = USB_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
InitData.EPInt = USB_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT, 8, NULL, 0);
USB_CDC_Add(&InitData);
USB_CDC_SetOnLineCoding(_OnLineCoding);
}

```

```

/*****
 *
 *      Public code
 *
 *****/

```

```

/*****
 *
 *      Get information that are used during enumeration
 */

```

```

/*****
 *
 *      USB_GetVendorId
 *
 *      Function description
 *      Returns vendor Id
 */
U16 USB_GetVendorId(void) {
    return 0x8765;
}

```

```

/*****
 *
 *      USB_GetProductId
 *
 *      Function description
 *      Returns product Id
 */
U16 USB_GetProductId(void) {
    return 0x1111;
}

```

```

/*****
 *
 *      USB_GetVendorName
 *
 *      Function description
 *      Returns vendor name
 */
const char * USB_GetVendorName(void) {
    return "Vendor";
}

```

```

/*****
 *
 *      USB_GetProductName
 *
 *      Function description
 *      Returns product name
 */
const char * USB_GetProductName(void) {
    return "CDC device";
}

```

```

/*****
 *
 *      USB_GetSerialNumber
 *
 *  Function description
 *  Returns serial number
 */
const char * USB_GetSerialNumber(void) {
    return "13245678";
}

/*****
 *
 *      MainTask
 *
 *  USB handling task.
 *  Modify to implement the desired protocol
 */
void MainTask(void);
void MainTask(void) {
    USB_Init();
    _AddCDC();
    USB_Start();
    while (1) {
        char ac[64];
        int  NumBytesReceived;
        //
        // Wait for configuration
        //
        while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        BSP_SetLED(0);
        //
        // Receive at maximum of 64 Bytes. If less data has been received,
        // this should be OK.
        //
        NumBytesReceived = USB_CDC_Receive(&ac[0], sizeof(ac));
        if (NumBytesReceived > 0) {
            USB_CDC_Write(&ac[0], NumBytesReceived);
        }
    }
}

```

7.3 Installing the driver

When the emUSB-CDC sample application is up and running and the target device is plugged into the computer's USB port Windows will detect the new hardware.



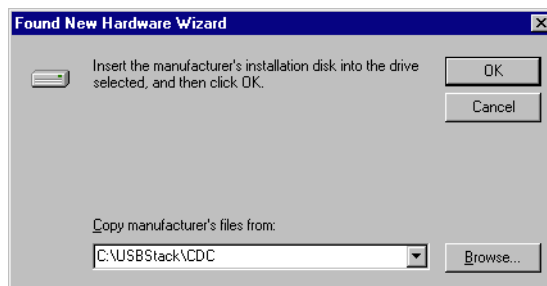
The wizard will ask you to help it find the correct driver files for the new device. First select the **Search for a suitable driver for my device (recommended)** option, then click the **Next** button.



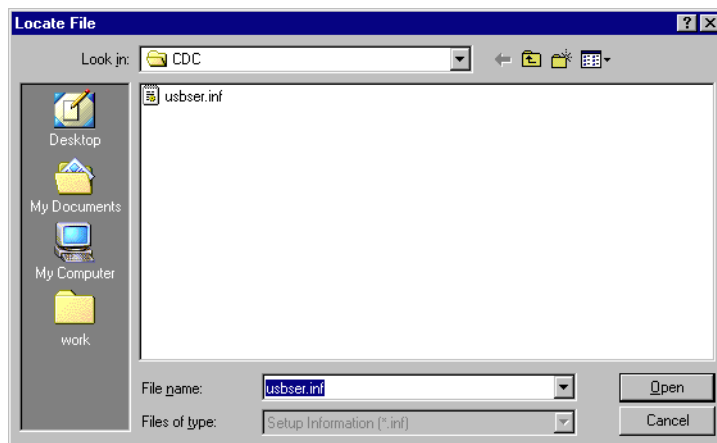
In the next step, you need to select the **Specify a location** option and click the **Next** button.



Click **Browse** to open the directory navigator.



Use the directory navigator to select `C:\USBStack\CDC` (or your chosen location) and click the **Open** button to select `usbser.inf`.



The wizard confirms your choice and starts to copy, when you click the **Next** button.



At this point, the installation is complete. Click the **Finish** button to dismiss the wizard.



7.3.1 The .inf file

The .inf file is required for installation.

It looks as follows:

```
;
; Device installation file for
; USB 2 COM port emulation
;
;
;
[Version]
Signature="$CHICAGO$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
DriverVer=01/08/2007,2.2.0.0
LayoutFile=Layout.inf

[Manufacturer]
%MFGNAME%=USB2SerialDeviceList

[USB2SerialDeviceList]
%USB2SERIAL%=USB2SerialInstall, USB\VID_8765&PID_0234

[DestinationDirs]
USB2SerialCopyFiles=12
DefaultDestDir=12

[USB2SerialInstall]
CopyFiles=USB2SerialCopyFiles
AddReg=USB2SerialAddReg

[USB2SerialCopyFiles]
usbser.sys,,0x20

[USB2SerialAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,, "MsPorts.dll,SerialPortPropPageProvider"

[USB2SerialInstall.Services]
AddService = usbser,0x0002,USB2SerialService

[USB2SerialService]
DisplayName = %USB2SERIAL_DISPLAY_NAME%
ServiceType = 1 ; SERVICE_KERNEL_DRIVER
StartType = 3 ; SERVICE_DEMAND_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\usbser.sys
LoadOrderGroup = Base

[Strings]
MFGNAME= "Manufacturer"
USB2SERIAL = "USB CDC serial port emulation"
USB2SERIAL_DISPLAY_NAME = "USB CDC serial port emulation"
```

red - required modifications

green - possible modifications

You have to personalize the .inf file on the red marked positions. Changes on the green marked positions are optional and not necessary for the correct function of the device.

Replace the red marked positions with your personal vendor Id (VID) and product Id (PID). These changes have to be identical with the modifications in the configuration file USB_Config.h to work correctly.

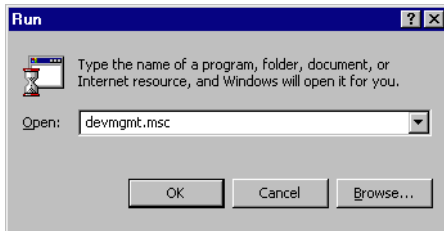
The required modifications of the file `USB_Conf.h` is described in the configuration chapter.

7.3.2 Installation verification

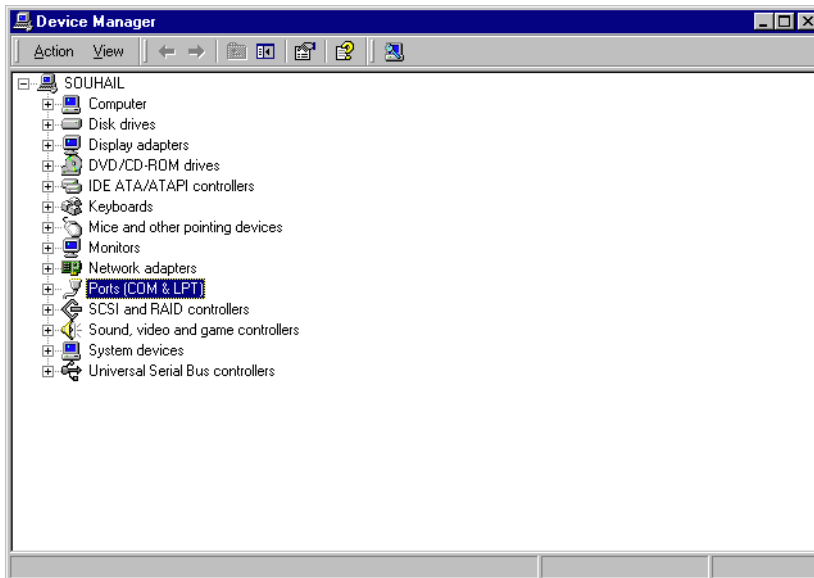
After the device has been installed, it can verify that the installation of the USB device was successful. Hence, take a look in the device manager to check that the USB device displayed.

The following steps perform:

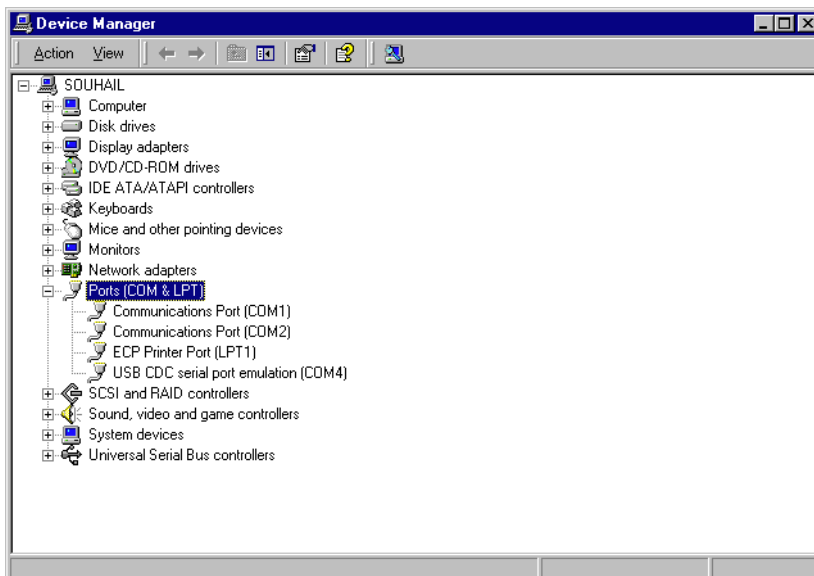
- Open the **Run** dialog box from the start menu.
Type `devmgmt.msc` and click **OK**:



- The **Device Manager** window is displayed and may look like this:



Click on the **Ports (COM & LPT)** branch to open the branch:



You should see the **USB CDC serial port emulation (COM_x)**, where _x gives the

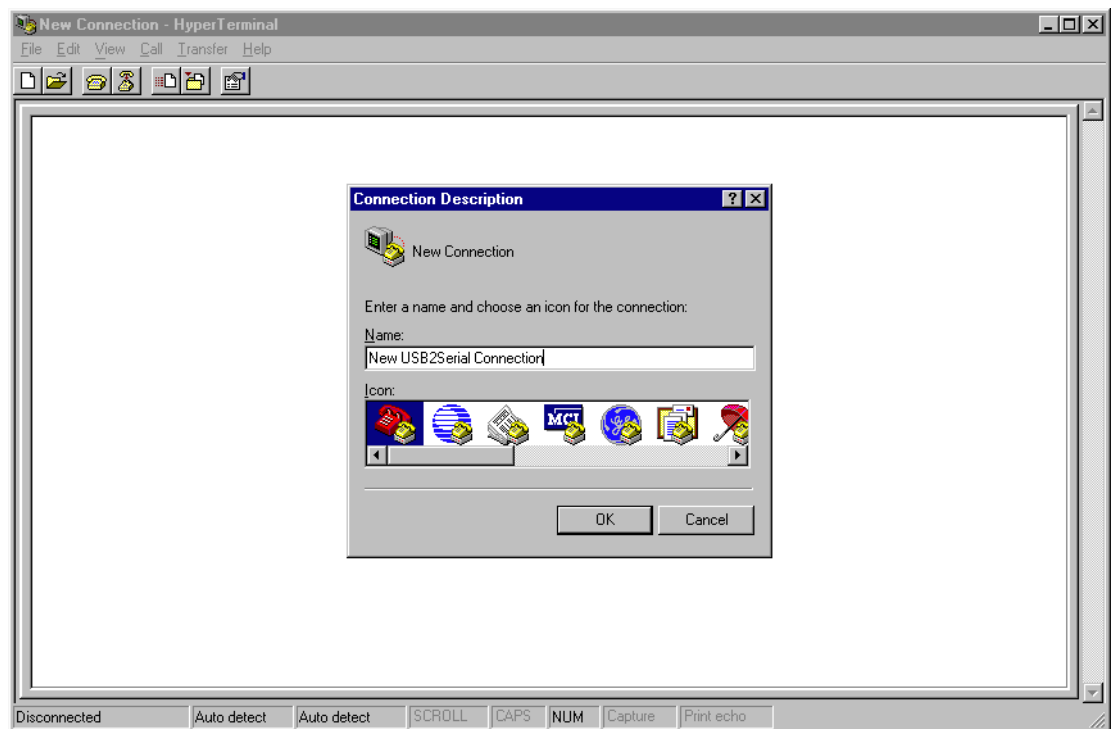
COM port number has Windows has assigned to the device.

7.3.3 Testing communication to the USB device

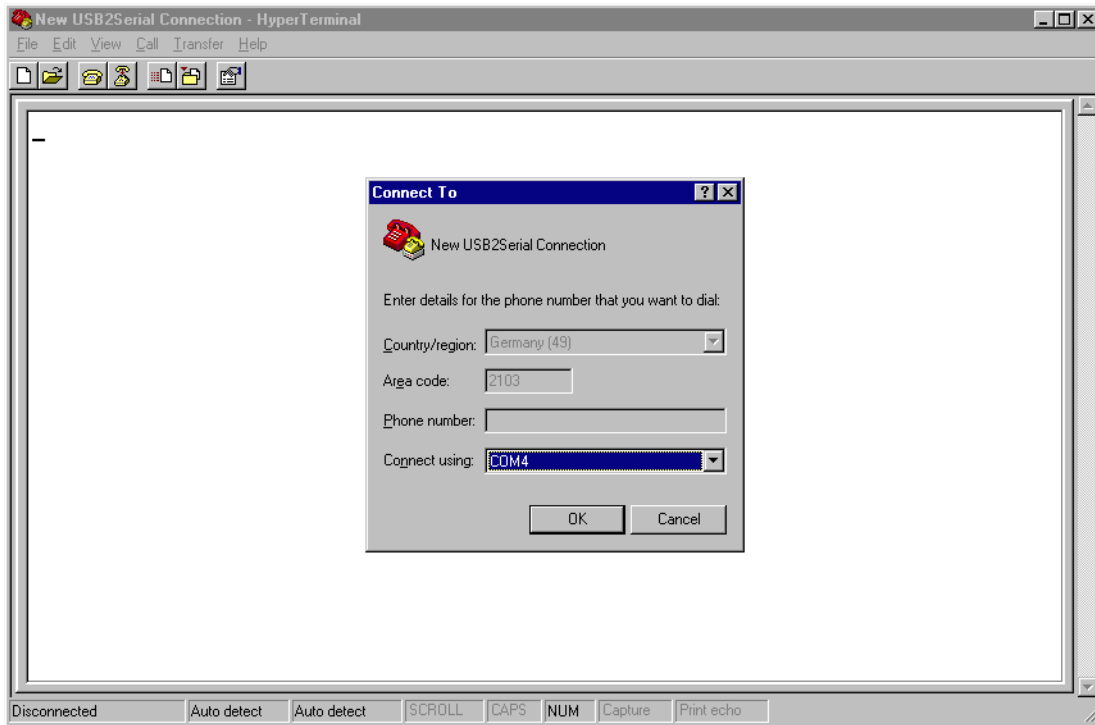
The start application is a simple echo server. This means each character that is entered and sent through the virtual serial port will be sent back by the USB device and will be shown by a terminal program. To test the communication to the device, a terminal program such as Hyperterminal, should be used.

This section shows how to check the communication between host and USB host using the Hyperterminal program.

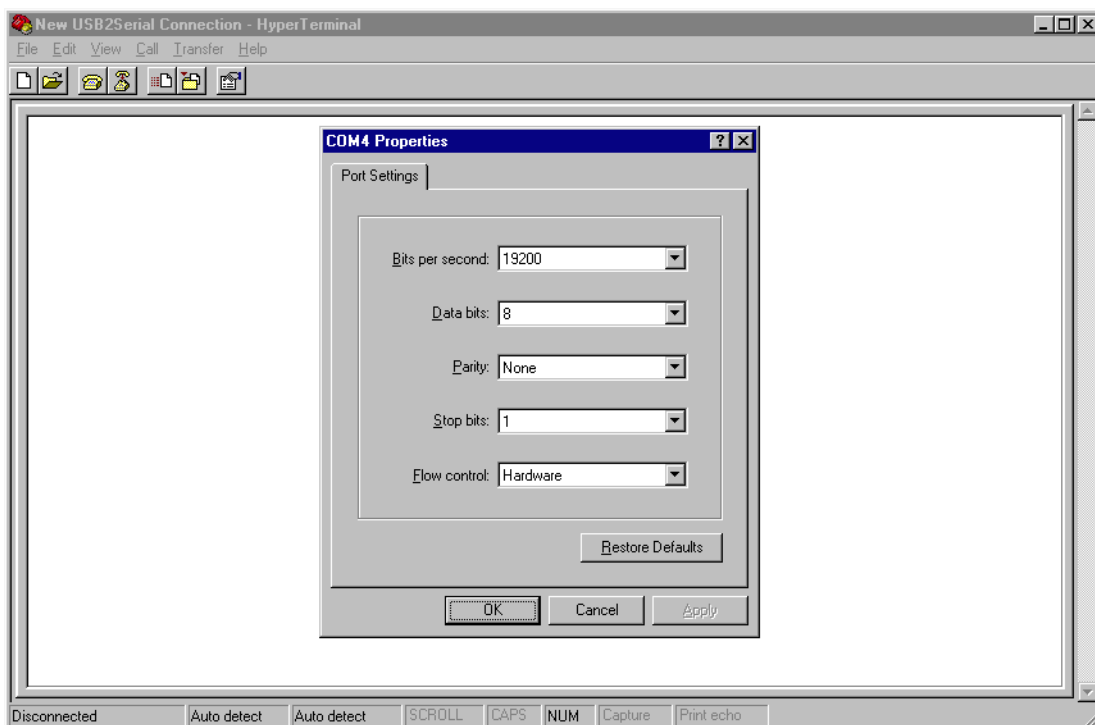
- Open the **Run** dialog box from the start menu.
Type `hypertrm.exe` and press Enter key to open the Hyperterminal.
Hyperterminal displays the Connection Description dialog.
Give this new connection a name as shown below and click **OK**.



- After creating the new connection, the **Connect To** dialog box is displayed and will ask which COM port you want to use. Click on the arrow for the **Connect Using** drop down box. Select **COMx**, where x is the port number that is assigned to your device by Windows. To confirm your choice click **OK**.

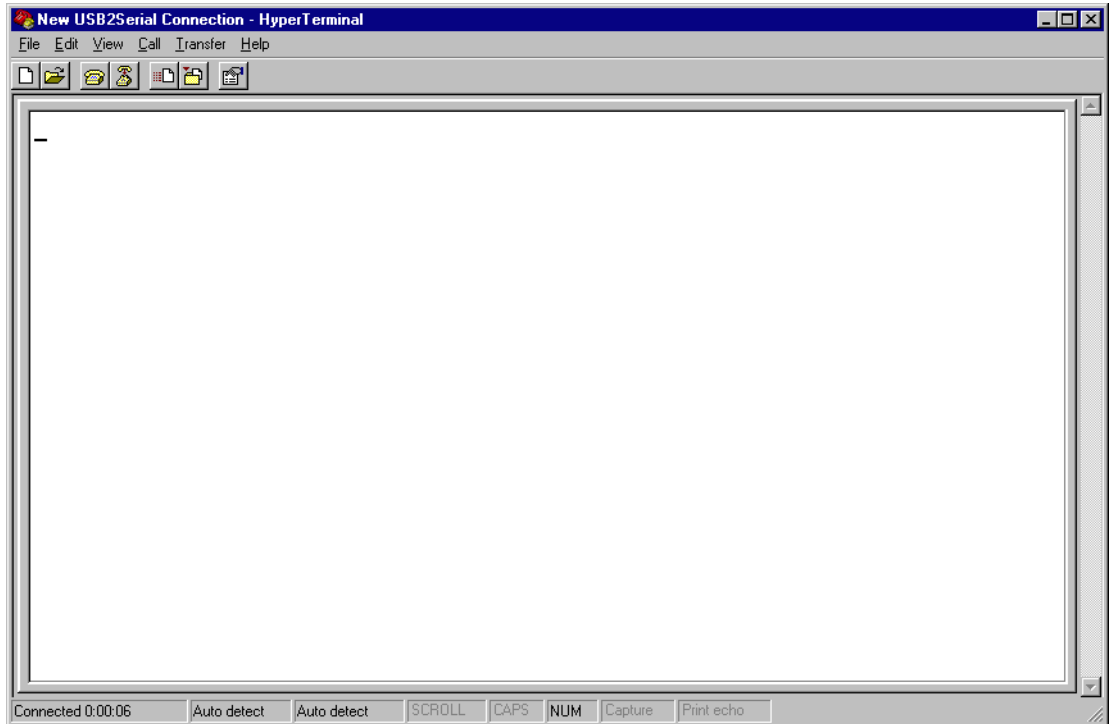


- The **COMx Property** dialog box is displayed to setup the connection properties. Setup the values as shown below:

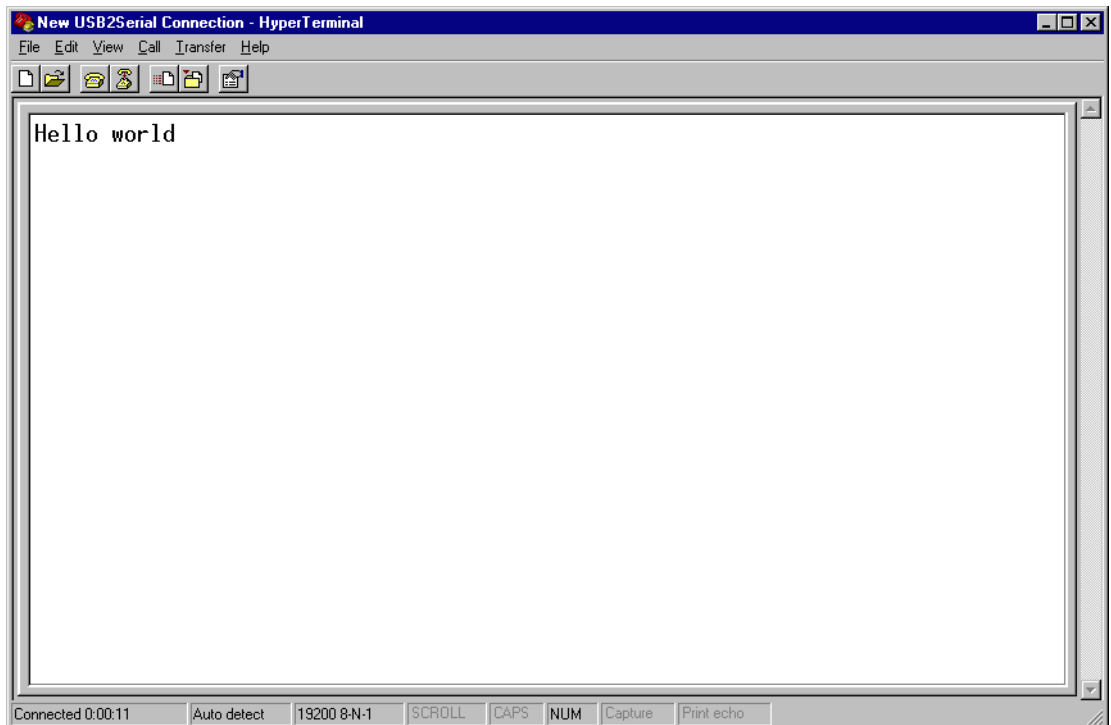


- To confirm your selection, click **OK**.

- Now everything is configured and an empty terminal window is shown.



Type any characters, these characters will be send to target. The echo of the target is shown in the terminal window:



7.4 Target API

This chapter describes the functions and data structures that can be used with the target application.

7.4.1 Interface function list

Name	Description
API functions	
<code>USB_CDC_Add()</code>	Adds CDC-class to the emUSB interface.
<code>USB_CDC_CancelRead()</code> <code>USB_CDC_CancelReadEx()</code>	Cancels an asynchronous read operation that is pending
<code>USB_CDC_CancelWrite()</code> <code>USB_CDC_CancelWriteEx()</code>	Cancels an asynchronous read operation that is pending
<code>USB_CDC_Read()</code> <code>USB_CDC_ReadEx()</code>	Reads data from host.
<code>USB_CDC_ReadOverlapped()</code> <code>USB_CDC_ReadOverlappedEx()</code>	Reads data from host asynchronously.
<code>USB_CDC_ReadTimed()</code> <code>USB_CDC_ReadTimedEx()</code>	Reads data from host with a given time-out.
<code>USB_CDC_Receive()</code> <code>USB_CDC_ReceiveEx()</code>	Reads data from host.
<code>USB_CDC_ReceiveTimed()</code> <code>USB_CDC_ReceiveTimedEx()</code>	Read data from host with a given time-out. This function returns immediately as soon as data has been received or a time-out occurs.
<code>USB_CDC_SetOnBreak()</code> <code>USB_CDC_SetOnBreakEx()</code>	Sets a callback for receiving a SEND_BREAK by the host.
<code>USB_CDC_SetOnLineCoding()</code> <code>USB_CDC_SetOnLineCodingEx()</code>	Sets a callback for registering changing of the "line-coding" by the host.
<code>USB_CDC_UpdateSerialState()</code> <code>USB_CDC_UpdateSerialStateEx()</code>	Changes the current serial state.
<code>USB_CDC_Write()</code> <code>USB_CDC_WriteEx()</code>	Writes data to host.
<code>USB_CDC_WriteOverlapped()</code> <code>USB_CDC_WriteOverlappedEx()</code>	Write data to host asynchronously.
<code>USB_CDC_WriteTimed()</code> <code>USB_CDC_WriteTimedEx()</code>	Writes data to the host with a given time-out.
<code>USB_CDC_WaitForRX()</code> <code>USB_CDC_WaitForRXEx()</code>	Waits for reading data transfer from the Host to be completed.
<code>USB_CDC_WaitForTX()</code> <code>USB_CDC_WaitForTXEx()</code>	Waits for writing data transfer to the Host to be completed.
<code>USB_CDC_WriteSerialState()</code> <code>USB_CDC_WriteSerialStateEx()</code>	Sends the current serial state to the Host.
Data structures	
<code>USB_CDC_INIT_DATA</code>	Initialization structure that is needed when adding an CDC interface.
<code>USB_CDC_ON_SET_BREAK</code>	Callback function to receive a break condition sent by the host.
<code>USB_CDC_ON_SET_LINE_CODING</code>	Callback registering line-coding changes.
<code>USB_CDC_LINE_CODING</code>	Structure that contains the new line-coding sent by the host.

Table 7.1: USB-CDC API overview

7.4.2 API functions

7.4.2.1 USB_CDC_Add()

Description

Adds CDC class to the USB interface.

Prototype

```
USB_CDC_HANDLE USB_CDC_Add(const USB_CDC_INIT_DATA * pInitData);
```

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_CDC_INIT_DATA</code> structure. For detailed information about the <code>USB_CDC_INIT_DATA</code> structure, refer to <i>USB_CDC_INIT_DATA</i> on page 206.

Table 7.2: USB_CDC_Add() parameter list

Return value

`== 0xFFFFFFFF` - New CDC Instance can not be created.

`!= 0xFFFFFFFF` - Handle to a valid CDC instance..

Additional information

After the initialization of general emUSB, this is the first function that needs to be called when the USB-CDC interface is used with emUSB. The returned value can be used with the CDC Ex-Function in order to talk to the right CDC instance.

For creating more more than one CDC-Instance please make sure the `USB_EnableIAD()` is called before, otherwise the CDC instances other than the first instance will work correctly.

7.4.2.2 USB_CDC_CancelRead() USB_CDC_CancelReadEx()

Description

Cancels a non-blocking read operation that is pending.

Prototype

```
void USB_CDC_CancelRead(void);  
void USB_CDC_CancelReadEx(USB_CDC_HANDLE hInst);
```

Additional information

This function should be called when a pending asynchronous read operation should be canceled. The function can be called from any task.

7.4.2.3 USB_CDC_CancelWrite() USB_CDC_CancelWriteEx()

Description

Cancels a non-blocking read operation that is pending.

Prototype

```
void USB_CDC_CancelWrite(void);
void USB_CDC_CancelWriteEx(USB_CDC_HANDLE hInst);;
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USB_CDC_Add().

Table 7.3: USB_CDC_CancelWriteEx() parameter list

Additional information

This function should be called when a pending asynchronously write operation should be canceled. It can be called from any task.

7.4.2.4 USB_CDC_Read() USB_CDC_ReadEx()

Description

Reads data from the host.

Prototype

```
int USB_CDC_Read(void * pData, unsigned NumBytes);  
int USB_CDC_ReadEx(USB_CDC_HANDLE hInst, void* pData, unsigned NumBytes);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USB_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 7.4: USB_CDC_Read()/USB_CDC_ReadEx()parameter list

Return value

Number of bytes that have been received.

7.4.2.5 USB_CDC_ReadOverlapped() USB_CDC_ReadOverlappedEx()

Description

Reads data from the host asynchronously.

Prototype

```
int USB_CDC_ReadOverlapped(void* pData, unsigned NumBytes);
int USB_CDC_ReadOverlappedEx(USB_CDC_HANDLE hInst,
                             void* pData,
                             unsigned NumBytes);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 7.5: USB_CDC_ReadOverlapped()/USB_CDC_ReadOverlappedEx() parameter list

Return value

Number of bytes that have already been received or have been copied from internal buffer.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, `USB_CDC_WaitForRX()` needs to be called.

7.4.2.6 USB_CDC_ReadTimed() USB_CDC_ReadTimedEx()

Description

Reads data from the host with a given time-out.

Prototype

```
int USB_CDC_ReadTimed(void* pData, unsigned NumBytes, unsigned ms);
int USB_CDC_ReadTimedEx(USB_CDC_HANDLE hInst, void* pData, unsigned
NumBytes, unsigned ms);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.
<code>ms</code>	Time-out given in milliseconds.

Table 7.6: USB_CDC_ReadTimed() parameter list

Return value

Number of bytes that have been read within the given time-out.

Additional information

This function blocks a task until all data have been read or a time-out occurs. This function also returns when target is disconnected from host or when a USB reset occurred.

7.4.2.7 USB_CDC_Receive() USB_CDC_ReceiveEx()

Description

Reads data from host.

Prototype

```
int USB_CDC_Receive(void * pBuffer, unsigned NumBytes);
int USB_CDC_ReceiveEx(USB_CDC_HANDLE hInst, void * pBuffer, unsigned
NumBytes);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pBuffer</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.

Table 7.7: USB_CDC_Receive() parameter list

Return value

Number of bytes that have been read.

Additional information

If no error occurs, this function returns the number of bytes received. If the connection has been closed, the return value is zero. Calling `USB_CDC_Receive()` will return as much data as is currently available up to the size of the buffer specified.

7.4.2.8 USB_CDC_ReceiveTimed() USB_CDC_ReceiveTimedEx()

Description

Reads data from host with a given time-out.

Prototype

```
int USB_CDC_ReceiveTimed(void * pBuffer, unsigned NumBytes, unsigned ms);
int USB_CDC_ReceiveTimedEx(USB_CDC_HANDLE hInst, void * pBuffer, unsigned
NumBytes, unsigned ms);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pBuffer</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.
<code>ms</code>	Time-out given in milliseconds.

Table 7.8: USB_CDC_ReceiveTimed() parameter list

Return value

Number of bytes that have been read within the time-out.

Additional information

If no error occurs, this function returns the number of bytes received.

If the connection has been closed, the return value is zero.

Calling `USB_CDC_ReceiveTimed()` will return as much data as is currently available up to the size of the buffer specified within the specified time-out.

7.4.2.9 USB_CDC_SetOnBreak() USB_CDC_SetOnBreakEx()

Description

Sets a callback for receiving a SEND_BREAK by the host.

Prototype

```
void USB_CDC_SetOnBreak (USB_CDC_ON_SET_BREAK * pfOnBreak);
void USB_CDC_SetOnBreakEx(USB_CDC_HANDLE      hInst,
                          USB_CDC_ON_SET_BREAK * pfOnBreak);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
pf	Pointer to the callback function <code>USB_CDC_ON_SET_BREAK</code> . For detailed information about the <code>USB_CDC_ON_SET_BREAK</code> function pointer, refer to <code>USB_CDC_ON_SET_BREAK</code> on page 207.

Table 7.9: USB_CDC_SetLineCoding() parameter list

Additional information

This function is used to register a user callback which should notify the application when a break condition was sent by the host. Refer to `USB_CDC_ON_SET_BREAK` on page 207 for detailed information.

7.4.2.10 USB_CDC_SetOnLineCoding() USB_CDC_SetOnLineCodingEx()

Description

Sets a callback for registering changing of the “line-coding” by the host.

Prototype

```
void USB_CDC_SetOnLineCoding(USB_CDC_ON_SET_LINE_CODING * pf);
void USB_CDC_SetOnLineCodingEx(USB_CDC_HANDLE hInst,
                                USB_CDC_ON_SET_LINE_CODING * pf);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pf</code>	Pointer to the callback function <code>USB_CDC_ON_SET_LINE_CODING</code> . For detailed information about the <code>USB_CDC_ON_SET_LINE_CODING</code> function pointer, refer to <i>USB_CDC_ON_SET_LINE_CODING</i> on page 208.

Table 7.10: USB_CDC_SetLineCoding() parameter list

Additional information

This function is used to register a user callback which should notify the application that the host has changed the line codings. Refer to *USB_CDC_ON_SET_LINE_CODING* on page 208 for detailed information.

7.4.2.11 USB_CDC_UpdateSerialState() USB_CDC_UpdateSerialStateEx()

Description

Updates the control line state of the.

Prototype

```
void USB_CDC_UpdateSerialState(USB_CDC_SERIAL_STATE * pSerialState);
void USB_CDC_UpdateSerialStateEx(USB_CDC_HANDLE hInst, USB_CDC_SERIAL_STATE
* pSerialState);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pSerialState</code>	Pointer to the <code>USB_CDC_SERIAL_STATE</code> structure, refer to <code>USB_CDC_SERIAL_STATE</code> on page 210.

Table 7.11: USB_CDC_SetLineCoding() parameter list

Additional information

This function function updates the control line state internally. In order to inform the host about the serial state change, refer to the function `USB_CDC_WriteSerialState()` `USB_CDC_WriteSerialStateEx()` on page 205.

7.4.2.12 USB_CDC_Write() USB_CDC_WriteEx()

Description

Write data to the host.

Prototype

```
void USB_CDC_Write(const void* pData, unsigned NumBytes);  
void USB_CDC_WriteEx(USB_CDC_HANDLE hInst, const void* pData, unsigned  
NumBytes);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.

Table 7.12: USB_CDC_Write() parameter list

7.4.2.13 USB_CDC_WriteOverlapped() USB_CDC_WriteOverlappedEx()

Description

Write data to the host asynchronously.

Prototype

```
void USB_CDC_WriteOverlapped(const void* pData, unsigned NumBytes);
void USB_CDC_WriteOverlappedEx(USB_CDC_HANDLE hInst, const void* pData,
unsigned NumBytes);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USB_CDC_Add().
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.

Table 7.13: USB_CDC_WriteOverlapped() parameter list

Return value

Number of bytes that have already been sent to the HOST.

Additional information

This function will not block the calling task. The write transfer will be initiated and the function returns immediately. In order to synchronize, `USB_CDC_WaitForTX()` needs to be called.

7.4.2.14 USB_CDC_WriteTimed() USB_CDC_WriteTimedEx()

Description

Writes data to the host with a given time-out.

Prototype

```
int USB_CDC_WriteTimed(const void * pData, unsigned NumBytes, unsigned ms)
int USB_CDC_WriteTimedEx(USB_CDC_HANDLE hInst, const void * pData, unsigned
NumBytes, unsigned ms);
```

Parameter	Description
<code>hInst</code>	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .
<code>pData</code>	Pointer to a buffer where the received data will be stored.
<code>NumBytes</code>	Number of bytes to read.
<code>ms</code>	Time-out given in milliseconds.

Table 7.14: USB_CDC_ReadTimed() parameter list

Return value

Number of bytes that have been written within the given time-out.

Additional information

This function blocks a task until all data have been read or a time-out occurs. This function also returns when target is disconnected from host or when a USB reset occurred.

7.4.2.15 USB_CDC_WaitForRX() USB_CDC_WaitForRXEx()

Description

Waits for reading data transfer from the host to be completed.

Prototype

```
void USB_CDC_WaitForRX(void);
void USB_CDC_WaitForRXEx(USB_CDC_HANDLE hInst);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USB_CDC_Add().

Table 7.15: USB_CDC_WaitForRX()/USB_CDC_WaitForRXEx parameter list

Additional information

This function should be called in order to synchronize task with the read data transfer that previously initiated.

7.4.2.16 USB_CDC_WaitForTX() USB_CDC_WaitForTXEx()

Description

Waits for writing data transfer to the host to be completed.

Prototype

```
void USB_CDC_WaitForTX(void);  
void USB_CDC_WaitForTXEx(USB_CDC_HANDLE hInst);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by <code>USB_CDC_Add()</code> .

Table 7.16: USB_CDC_WaitForTX()/USB_CDC_WaitForTXEx parameter list

Additional information

This function should be called in order to synchronize task with the write data transfer that previously initiated.

7.4.2.17 USB_CDC_WriteSerialState() USB_CDC_WriteSerialStateEx()

Description

Sends the current control line serial state to the host.

Prototype

```
void USB_CDC_WriteSerialState(void);
void USB_CDC_WriteSerialStateEx(USB_CDC_HANDLE hInst);
```

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USB_CDC_Add().

Table 7.17: USB_CDC_WaitForTX()/USB_CDC_WaitForTXEx parameter list

Additional information

This function should be called in order to inform the host about the control serial state of the CDC instance. It may be called within the same function or in a another task dedicated for sending the serial state.

Please note the function is a blocking function, which means the function will return host has received the serial state.

7.4.3 Data structures

7.4.3.1 USB_CDC_INIT_DATA

Description

Initialization structure that is needed when adding an CDC interface to emUSB.

Prototype

```
typedef struct {  
    U8 EPIn;  
    U8 EPOut;  
    U8 EPInt;  
} USB_CDC_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host
EPOut	Endpoint for receiving data from the host
EPInt	Endpoint for sending status information.

Table 7.18: USB_CDC_INIT_DATA elements

Additional Information

This structure is used when the CDC interface is added to emUSB. [EPInt](#) is in this version of the emUSB CDC component not used, status information are not sent to the host.

7.4.3.2 USB_CDC_ON_SET_BREAK

Description

Callback function to receive a break condition sent by the host.

Prototype

```
typedef void USB_CDC_ON_SET_BREAK(unsigned BreakDuration);
```

Member	Description
<code>BreakDuration</code>	The BreakDuration give the length of time, in milliseconds, of the break signal.

Table 7.19: USB_CDC_ON_SET_LINE_CODING elements

Additional Information

This type of callback is used to notify the application that the host has sent a break condition. If `BreakDuration` is `0xFFFF`, then the host will send a break until another `SendBreak` request is received with `BreakDuration` of `0x0000`.

Note: Since the callback is mostly called within an interrupt service routine, this callback should set any variables/events that signals any events that need to be done.

7.4.3.3 USB_CDC_ON_SET_LINE_CODING

Description

Callback function to register line-coding changes.

Prototype

```
typedef void USB_CDC_ON_SET_LINE_CODING(USB_CDC_LINE_CODING * pLineCoding);
```

Member	Description
pLineCoding	Pointer to <code>USB_CDC_LINE_CODING</code> structure

Table 7.20: USB_CDC_ON_SET_LINE_CODING elements

Additional Information

This type of callback is used to notify the application that the host has change the line codings. For example the baud rate has been changed. The new "line-coding" are passed through the structure `USB_CD_LINE_CODING`. Refer to *USB_CDC_LINE_CODING* on page 209 for more information about the elements of these structure.

Note: Since the callback is mostly called within an interrupt service routine, this callback should set any variables/events that signals any events that need to be done.

7.4.3.4 USB_CDC_LINE_CODING

Description

Structure that contains the new line-coding sent by the host.

Prototype

```
typedef struct {
    U32 DTERate;
    U8 CharFormat;
    U8 ParityType;
    U8 DataBits;
} USB_CDC_LINE_CODING;
```

Member	Description
DTERate	The data transfer rate for the device in bits per second.
CharFormat	Contain the stop bits: 0 - 1 Stop bit 1 - 1.5 Stop bits 2 - 2 Stop bits
ParityType	Specifies the parity type: 0 - None 1 - Odd 2 - Even 3 - Mark 4 - Space
DataBits	Specifies the bits per byte: (5, 6, 7, 8, 16)

Table 7.21: USB_CDC_LINE_CODING elements

7.4.3.5 USB_CDC_SERIAL_STATE

Description

Structure that contains the new line-coding sent by the host.

Prototype

```
typedef struct {
    U8 DCD;
    U8 DSR;
    U8 Break;
    U8 Ring;
    U8 FramingError;
    U8 ParityError;
    U8 OverRunError;
    U8 CTS;
} USB_CDC_SERIAL_STATE;
```

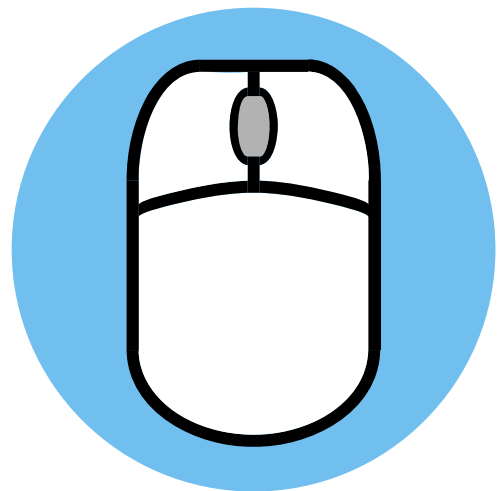
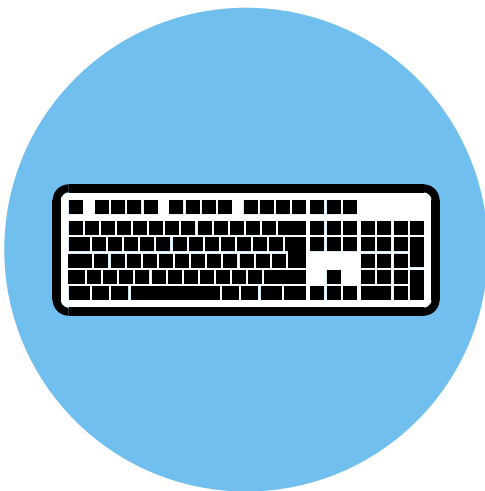
Member	Description
DCD	Data Carrier Detect: Tells that the device is connected to the telephone line.
DSR	Data Set Read: Device is ready to receive data.
Break	
Ring	Device indicates that it has detected a ring signal on the telephone line.
FramingError	When set to 1, the device indicates a framing error.
ParityError	When set to 1, the device indicates an parity error.
OverRunError	When set to 1, the device indicates an over-run error.
CTS	Clear to send: Acknowledges RTS and allows the host to transmit.

Table 7.22: USB_CDC_LINE_CODING elements

Chapter 8

Human Interface Device Class (HID)

This chapter gives a general overview of the HID class and describes how to get the HID component running on the target.



8.1 Overview

The Human Interface Device class (HID) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol was defined for the handling of devices which are used by humans to control the operation of computer systems.

An installation of a custom-host USB driver is not necessary, because the USB human interface device class is standardized and every major OS already provides host drivers for it.

Method of communication

HID always uses interrupt end points. Since interrupt endpoints are limited to at most one packet of at most 64 bytes per frame (on full speed devices), the transfer rate is limited to 64000 bytes/sec, in reality much less than that.

8.1.1 Further reading

The following documents define the HID class and have been used to implement and verify the HID component:

- [HID1]
Device Class Definition for Human Interface Devices (HID), Firmware Specification—6/27/01 Version 1.11
- [HID2]
HID Usage Tables, 1/21/2005 Version 1.12

8.1.2 Categories

Devices which are in the HID class basically fall into one of 2 categories:

“True HIDs” and “vendor specific HIDs”, explained in the following. One or more examples for both categories are provided.

8.1.2.1 “True HIDs”

HID devices communicating with the host operating system. Devices which are used by a human being to input / output data, but do not directly exchange data with an application program running on the host.

Typical examples

- Keyboard
- Mouse and similar pointing devices
- Joystick
- Game pad
- Front-panel controls - for example, switches and buttons.

8.1.2.2 “Vendor specific HIDs”

These are HID devices communicating with an application program. The host OS loads the same driver it loads for any “true HID” and will automatically enumerate the device, but it can not communicate with the device. When analyzing the report descriptor, the host finds out that it can not exchange information with the device; the device uses a protocol which is meaningless to the HID driver of the host. The Host will therefore not exchange information with the device. A host recognizes a vendor specific HID by its vendor defined usage page in the report descriptor: The numerical value of the usage page is between 0xFF00 and 0xFFFF.

An application has the chance to communicate with the particular device using API functions offered by the host. This allows an application program to communicate with the device without having to load a driver. HID does not take advantage of the full USB bus bandwidth; bulk communication can be much faster, but requires a driver. Therefore it can be a good choice to select HID as device class, especially if easy-of use is important and high communication speed is not a requirement.

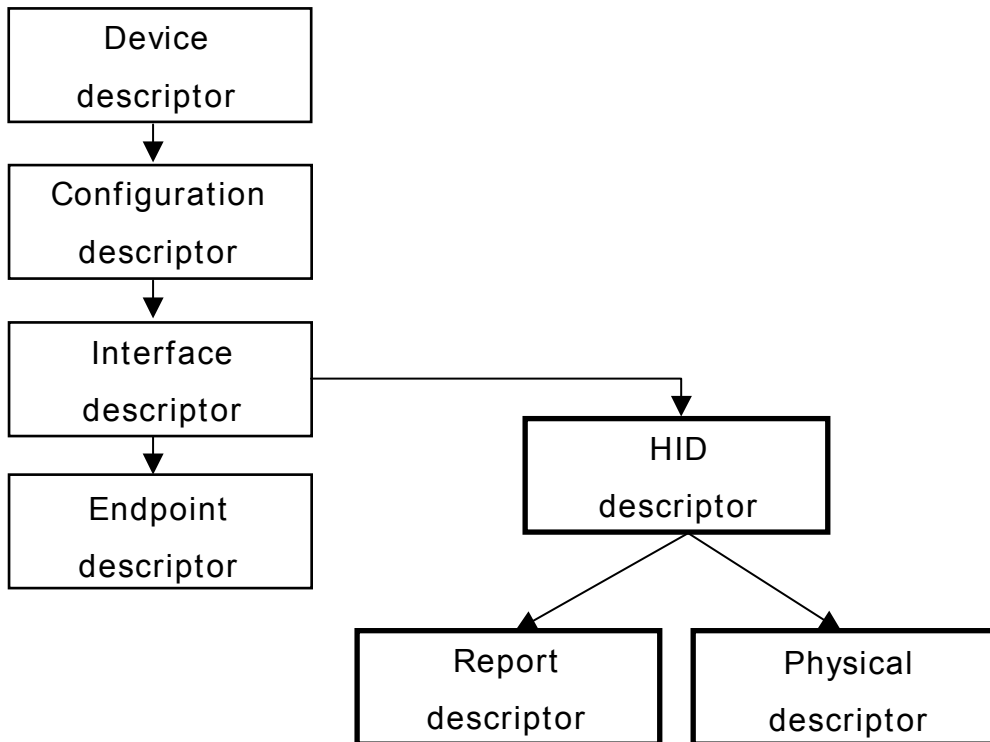
Typical examples

- Bar-code reader
- Thermometer
- Voltmeter
- Low-speed JTAG emulator
- UPS (Uninterruptible power supply)

8.2 Background information

8.2.1 HID descriptors

This section gives an overview about the HID class specific descriptors. The HID descriptors are defined in the *Device Class Definition for Human Interface Devices (HID)* of the USB Implementers Forum. Refer to the USB Implementers Forum website, www.usb.org, for detailed information about the USB HID standard.



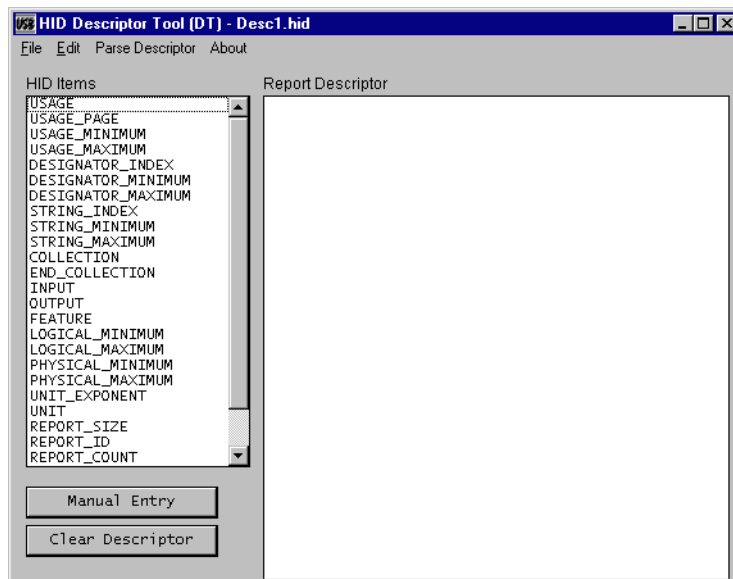
8.2.1.1 HID descriptor

A HID descriptor contains the report and optional the physical descriptors. It specifies the number, type, and size of report and physical descriptors.

8.2.1.2 Report descriptor

The data exchanged between host and device is exchanged in so called "reports". The report descriptor defines the format of a report. In general, HIDs require a Report descriptor as defined in the *Device Class Definition for Human Interface Devices (HID)*. The only exception to this are very basic HIDs such as mouse or keyboards. This implementation of HID always requires a report descriptor.

The USB Implementers Forum provides an application which helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from: <http://www.usb.org/developers/hidpage/>



8.2.1.3 Physical descriptor

Physical descriptor sets are optional descriptors which provide information about the part or parts of the human body used to activate the controls on a device. Physical descriptors are not currently supported.

8.3 Configuration

8.3.1 Initial configuration

To get emUSB up and running as well as doing an initial test, the configuration as it is delivered should not be modified. The configuration must only be modified, if emUSB should be used in your final product. Refer to the section *Configuration* on page 41 to get detailed information about the functions which has to be adapted before you can release a final product version.

8.3.2 Final configuration

Generating a report descriptor

This step is only required if your product is an vendor specific human interface device. The report descriptor provided in the example application can typically be used without any modification. The vendor defined usage page should be adapted in a final product. Vendor defined usage pages can be in the range from 0xFF00 - 0xFFFF. The low byte can be selected by the application programmer. It needs to be identical on both target and host and should be unique (as unique as an 8-bit value can be). The example(s) use the value 0x12; this value is defined at the top of the application program with the macro `USB_HID_DEFAULT_VENDOR_PAGE`.

8.4 Example application

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

The following start application files are provided:

File	Description
HID_Mouse.c	Simple mouse example. ("True HID" example)
HID_Echo1.c	Modified echo server. ("vendor specific" example)

Table 8.1: Supplied example HID applications

8.4.1 HID_Mouse.c

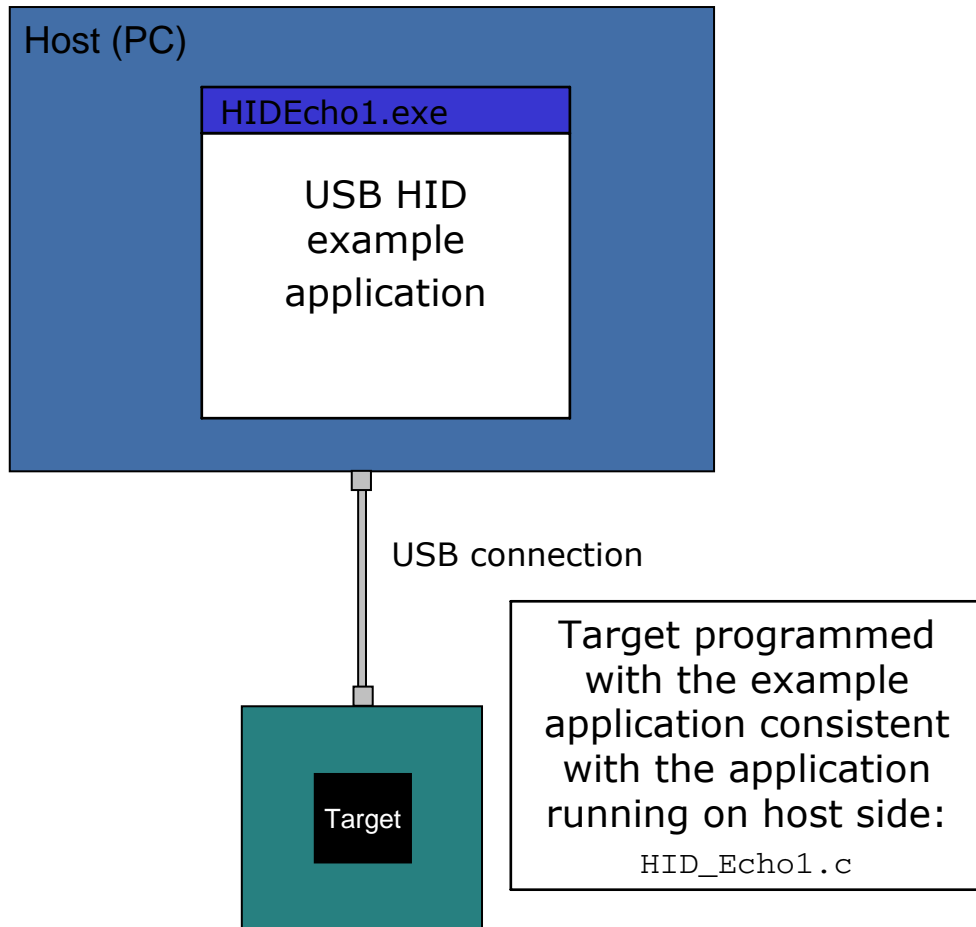
HID_Mouse.c is a typical example for a "true HID" implementation. The host identifies the device which is programmed with this example as a mouse. After the device is enumerated moves the mouse cursor in an endless loop to the left and after a short delay back to the right.

8.4.1.1 Running the example

1. Add HID_Mouse.c to your project and build and download the application into the target.
2. When you connect your target to the host via USB, Windows will detect the new HID device.
3. If a connection can be established moves the mouse cursor in an endless loop to the left and after a short delay back to the right as long as you do not disconnect your target.

8.4.2 HID_Echo1.c

`HID_Echo1.c` is a typical example for a “vendor specific HID” implementation. The HID start application (`HID_Echo1.c` located in the `Application` subfolder) is a modified echo server; the application receives data byte by byte, increments every single byte and sends it back to the host.

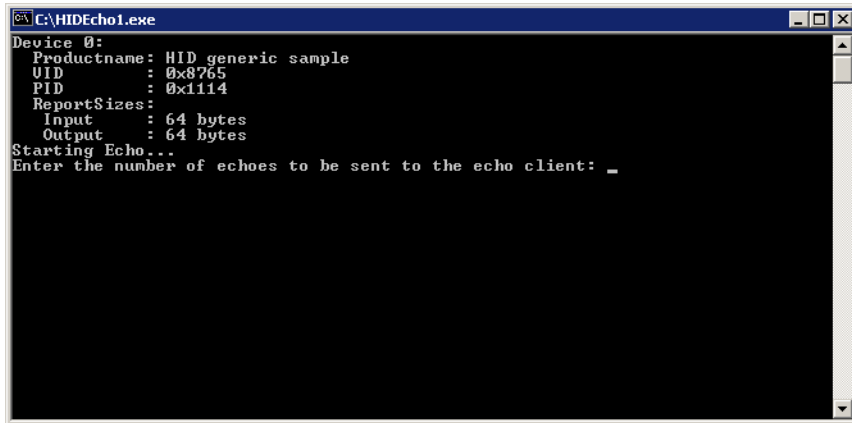


To use this application, include the source code file `HID_Echo1.c` into your project and compile and download it into your target. Run `HIDEcho1.exe` after target is connected to host and the enumeration process has been completed. The PC application is supplied as executable in the `HID\SampleApp\Exe` directory. The source code of the PC example is also supplied. Refer to section *Compiling the PC example application* on page 219 for more information to the PC example project.

8.4.2.1 Running the example

1. Add `HID_Echo1.c` to your project and build and download the application into the target.
2. Connect your target to the host via USB while the example application is running, Windows will detect the new HID device.
3. If a connection can be established, it exchanges data with the target, testing the USB connection. If the host example application can communicate with the emUSB device, the example application outputs the product name, vendor and product id and the report size which will be used to communicate with the target. The target will be in interactive mode.

Example output of `HID_Echo1.exe`:

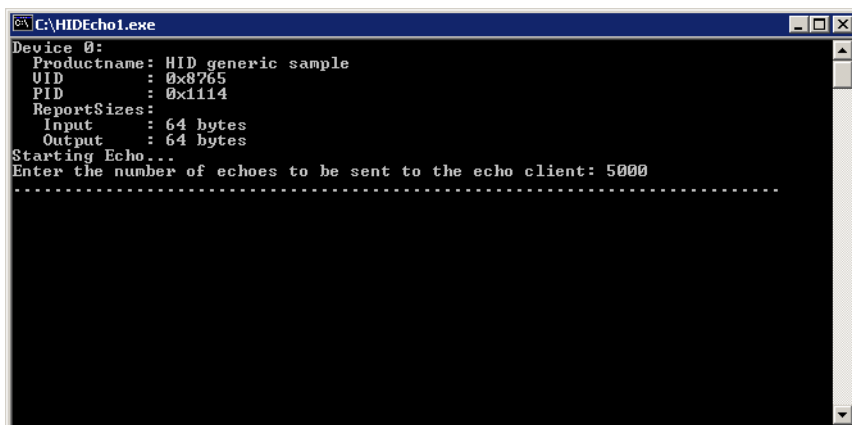


```

C:\HIDEcho1.exe
Device 0:
Productname: HID generic sample
UID       : 0x8765
PID       : 0x1114
ReportSizes:
  Input   : 64 bytes
  Output  : 64 bytes
Starting Echo...
Enter the number of echoes to be sent to the echo client: _

```

4. Enter the number of reports that should be transmitted, when the device is connect. Every dot in the terminal window indicates a transmission.



```

C:\HIDEcho1.exe
Device 0:
Productname: HID generic sample
UID       : 0x8765
PID       : 0x1114
ReportSizes:
  Input   : 64 bytes
  Output  : 64 bytes
Starting Echo...
Enter the number of echoes to be sent to the echo client: 5000
.....

```

8.4.2.2 Compiling the PC example application

To compile the example application you need a Microsoft compiler. The compiler is part of Microsoft Visual C++ 6.0 or Microsoft Visual Studio .Net. The source code of the example application is located in the subfolder `HID\SampleApp`. Open the file `USBHID_Start.dsw` and compile the source choose **Build | Build SampleApp.exe** (Shortcut: F7). To run the executable choose **Build | Execute SampleApp.exe** (Shortcut: CTRL-F5).

Note: The Microsoft Windows Driver Development Kit (DDK) is required to compile the HID host example application. Refer to <http://www.microsoft.com/whdc/dev-tools/ddk/default.msp> for more information.

8.5 Target API

This section describes the functions that can be used on the target system.

General information

To communicate with the host, the example application project includes USB-specific header and source files (`USB.h`, `USB_Main.c`, `USB_Private.c`, `USB_Read.c`, `USB_Setup.c`, `USB_Write.c`). These files contain API functions to communicate with the USB host.

Purpose of the USB Device API functions

To have an easy start up when writing an application on the device side, these API functions have a simple interface and handle all operations that need to be done to communicate with the host.

Therefore, all operations that need to write to or read from the emUSB are handled internally by the provided API functions.

8.5.1 Target interface function list

Function	Description
API functions	
<code>USB_HID_Add()</code>	Adds HID-class to the emUSB interface.
<code>USB_HID_Read()</code>	Reads data from host.
<code>USB_HID_Write()</code>	Write data to host.
Data structures	
<code>USB_HID_INIT_DATA</code>	Initialization structure that is required when adding an HID interface.

Table 8.2: USB-HID target interface function list

8.5.2 USB-HID functions

8.5.2.1 USB_HID_Add()

Description

Adds HID class device to the USB interface.

Prototype

```
void USB_HID_Add(const USB_HID_INIT_DATA * pInitData);
```

Parameter	Description
<code>pInitData</code>	Pointer to a <code>USB_HID_INIT_DATA</code> structure. For detailed information about the <code>USB_HID_INIT_DATA</code> structure, refer to <i>USB_HID_INIT_DATA</i> on page 224.

Table 8.3: USB_HID_Add() parameter list

Additional information

After the initialization of general emUSB, this is the first function that needs to be called when the USB-HID interface is used with emUSB.

8.5.2.2 USB_HID_Read()

Description

Reads data from the host.

Prototype

```
int USB_HID_Read(void* pData, unsigned NumBytes);
```

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Table 8.4: USB_HID_Read() parameter list

Return value

Number of bytes that have been received.

8.5.2.3 USB_HID_Write()

Description

Writes data to the host.

Prototype

```
void USB_HID_Write(const void* pData, unsigned NumBytes);
```

Parameter	Description
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to write.

Table 8.5: USB_HID_Write() parameter list

8.5.3 Data structures

8.5.3.1 USB_HID_INIT_DATA

Description

Initialization structure that is needed when adding a CDC interface to emUSB.

Prototype

```
typedef struct {
    U8 EPIn;
    U8 EPOut;
    const U8 * pReport;
    U16 NumBytesReport;
} USB_HID_INIT_DATA;
```

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pReport	Pointer to a report descriptor.
NumBytesReport	Size of the HID report.

Table 8.6: USB_HID_INIT_DATA elements

Additional Information

This structure is used when the HID interface is added to emUSB. [EPOut](#) is not required.

[pReport](#) points to a report descriptor. A report descriptor is a structure which is used to transmit HID control data to and from a human interface device. A report descriptor defines the format of a report. It is composed of report items that define one or more top-level collections. Each collection defines one or more HID reports.

Refer to *Universal Serial Bus Specification, 1.0 Version* and the latest version of the *HID Usage Tables* guide to get detailed information about the HID input, output and feature reports.

The USB Implementers Forum provide an application that helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from: <http://www.usb.org/developers/hidpage/>.

The report descriptor used in the supplied example application `HID_Echo1.c` should match to the requirements of most "vendor specific HID" applications. The report size is defined to 64 bytes. As mentioned before, interrupt endpoints are limited to at most one packet of at most 64 bytes per frame (on full speed devices), so that the defined report size is exhausted.

Example

Example excerpt from `HID_Mouse.c`:

```
static void _AddHID(void) {
    USB_HID_INIT_DATA InitData;
    U8 Interval = 10;
    static U8 acBuffer[64];

    memset(&InitData, 0, sizeof(InitData));
    InitData.EPIn = USB_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT, Interval, NULL, 0);
    // Note: Next line is optional. EPOut is not required!
    InitData.EPOut = USB_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_INT, Interval, /
        &acBuffer[0], sizeof(acBuffer));

    InitData.pReport = _aHIDReport;
    InitData.NumBytesReport = sizeof(_aHIDReport);
    USB_HID_Add(&InitData);
}
```

8.6 Host API

This chapter describes the functions that can be used with the Windows host system. These functions are only required if use the emUSB-HID component to design a vendor specific HID.

General information

To communicate with the target USB-HID stack, the example application project includes a USB-HID specific source and header file (`USBHID.c`, `USBHID.h`). These files contain API functions to communicate with the USB-HID target through the USB-Bulk driver.

Purpose of the USB Host API functions

To have an easy start-up when writing an application on the host side, these API functions have simple interfaces and handle all operations that need to be done to communicate with the target USB-HID stack.

8.6.1 Host API function list

Function	Description
API functions	
<code>USBHID_Close()</code>	Closes the connection an open device.
<code>USBHID_Open()</code>	Opens a handle to the device.
<code>USBHID_Init()</code>	Initializes the USB human interface device.
<code>USBHID_Exit()</code>	Closes the connection an open device.
<code>USBHID_GetNumAvailableDevices()</code>	Returns the number of available devices.
<code>USBHID_GetProductName()</code>	Returns the product name.
<code>USBHID_GetInputReportSize()</code>	Returns the input report size of the device.
<code>USBHID_GetOutputReportSize()</code>	Returns the output report size of the device.
<code>USBHID_GetProductId()</code>	Returns the product Id of the device.
<code>USBHID_GetVendorId()</code>	Returns the vendor id of the device.
<code>USBHID_RefreshList()</code>	Refreshes connection info list.
<code>USBHID_SetVendorPage()</code>	Sets the vendor page.

Table 8.7: USB-HID host interface function list

8.6.2 USB-HID functions

8.6.2.1 USBHID_Close()

Description

Closes the connection an open device.

Prototype

```
void USBHID_Close (unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code>

Table 8.8: USBHID_Close() parameter list

8.6.2.2 USBHID_Open()

Description

Opens a handle to the device that should be opened.

Prototype

```
int USBHID_Open (unsigned Id)
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .

Table 8.9: USBHID_Open() parameter list

Return value

== 0: Opening was successful or already opened.

== 1: Error. Handle to the device could not opened.

8.6.2.3 USBHID_Init()

Description

Sets the specific vendor page, initialize the USB HID User API and retrieve the information of the HID device.

Prototype

```
void USBHID_Init(U8 VendorPage);
```

Parameter	Description
VendorPage	This parameter specifies the lower 8 bits of the vendor specific usage page number. It must be identical on both device and host.

Table 8.10: USBHID_Init() parameter list

8.6.2.4 USBHID_Exit()

Description

Closes the connection an open device.

Prototype

```
void USBHID_Exit(void);
```

8.6.2.5 USBHID_GetNumAvailableDevices()

Description

Returns the number of the available devices.

Prototype

```
unsigned USBHID_GetNumAvailableDevices(U32 * pMask);
```

Parameter	Description
<code>pMask</code>	Pointer to unsigned integer value which is used to store the bit mask of available devices. This parameter may be <code>NULL</code> .

Table 8.11: USBHID_GetNumAvailableDevices() parameter list

Return value

Returns the number of available devices.

Additional information

`pMask` will be filled by this routine. It should be interpreted as bit mask where a bit set means this device is available. For example, Device 0 and device 2 are available, if `pMask` has the value 0x00000005.

8.6.2.6 USBHID_GetProductName()

Description

Stores the name of the device into `pBuffer`.

Prototype

```
int USBHID_GetProductName(unsigned Id, char * pBuffer, unsigned NumBytes);
```

Parameter	Description
<code>DeviceIndex</code>	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .
<code>pBuffer</code>	Pointer to a buffer for the product name.
<code>NumBytes</code>	Size of the <code>pBuffer</code> in bytes.

Table 8.12: USBHID_GetProductName() parameter list

Return value

== 0: On error.

== 1: On success.

8.6.2.7 USBHID_GetInputReportSize()

Description

Returns the input report size of the device.

Prototype

```
int USBHID_GetInputReportSize(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .

Table 8.13: USBHID_GetInputReportSize() parameter list

Return value

== 0: On error.

<> 1: Size of the report in bytes.

8.6.2.8 USBHID_GetOutputReportSize()

Description

Returns the output report size of the device.

Prototype

```
int USBHID_GetOutputReportSize(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .

Table 8.14: USBHID_GetOutputReportSize() parameter list

Return value

== 0: On error.

<> 1: Size of the report in bytes.

8.6.2.9 USBHID_GetProductId()

Description

Returns the product Id of the device.

Prototype

```
U16 USBHID_GetProductId(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .

Table 8.15: USBHID_GetProductId() parameter list

Return value

== 0: On error.

<> 1: Product Id.

8.6.2.10 USBHID_GetVendorId()

Description

Returns the vendor Id of the device.

Prototype

```
U16 USBHID_GetVendorId(unsigned Id);
```

Parameter	Description
DeviceIndex	Index of the HID device. This is the bit number of the mask returned by <code>USBHID_GetNumDevices()</code> .

Table 8.16: USBHID_GetVendorId() parameter list

Return value

== 0: On error.

<> 1: Vendor Id.

8.6.2.11 USBHID_RefreshList()

Description

Refreshes the connection info list.

Prototype

```
void USBHID_RefreshList(void);
```

Additional information

Note, that any open handle to the device will be closed while refreshing the connection info list.

8.6.2.12 USBHID_SetVendorPage()

Description

Sets the vendor page so that all HID device with the specified page will be found.

Prototype

```
void USBHID_SetVendorPage(U8 VendorPage);
```

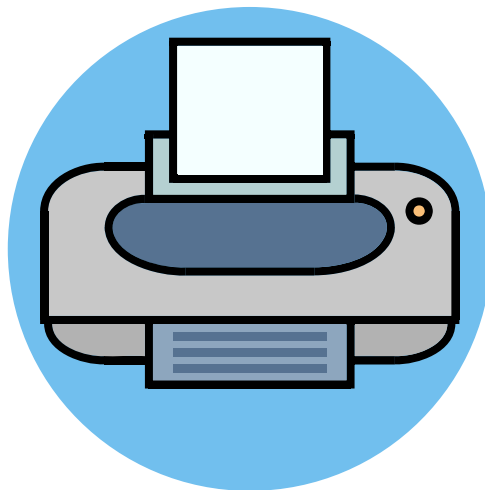
Parameter	Description
VendorPage	This parameter specifies the lower 8 bits of the vendor specific usage page number. It must be identical on both device and host.

Table 8.17: USBHID_SetVendorPage() parameter list

Chapter 9

Printer Class

This chapter describes how to get emUSB up and running as a printer device.



9.1 Overview

The Printer Class is an abstract USB class protocol defined by the USB Implementers Forum. This protocol delivers the existing printing command-sets to a printer over USB.

9.1.1 Configuration

The configuration section will later on be modified to match the real application. For the purpose of getting emUSB up and running as well as doing an initial test, the configuration as delivered should not be modified.

9.2 The example application

The start application (in the `Application` subfolder) is a simple data sink, which can be used to test emUSB. The application receives data bytes from the host which it displays in the terminal I/O window of the debugger.

Source code of `USB_Printer.c`:

```

/*****
 *          SEGGER MICROCONTROLLER SYSTEME GmbH
 *          Solutions for real time microcontroller applications
 *****/
 *
 *          (C) 2003 - 2007  SEGGER Microcontroller Systeme GmbH
 *
 *          www.segger.com    Support: support@segger.com
 *
 *****/
 *
 *          USB device stack for embedded applications
 *
 *****/
-----
File      : USB_PrinterClass.c
Purpose   : Sample implementation of USB printer device class
-----Literature-----
Universal Serial Bus Device Class Definition for Printing Devices
Version 1.1 January 2000
-----  END-OF-HEADER  -----
*/

#include <stdio.h>
#include <string.h>
#include "USB_PrinterClass.h"
#include "BSP.h"

/*****
 *
 *          static data
 *
 *****/
static U8 _acData[512];

/*****
 *
 *          static code
 *
 *****/

/*****
 *
 *          _GetDeviceIdString
 *
 */
static const char * _GetDeviceIdString(void) {
    const char * s = "CLASS:PRINTER;MODEL:HP LaserJet 6MP;"
                    "MANUFACTURER:Hewlett-Packard;"
                    "DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;"
                    "COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;";

    return s;
}

/*****
 *
 *          _GetHasNoError
 *
 */
static U8 _GetHasNoError(void) {
    return 1;
}

/*****
 *
 *          _GetIsSelected
 *
 */
static U8 _GetIsSelected(void) {
    return 1;
}

```

```

}

/*****
 *
 *      _GetIsPaperEmpty
 *
 */
static U8 _GetIsPaperEmpty(void) {
    return 0;
}

/*****
 *
 *      _OnDataReceived
 *
 */
static int _OnDataReceived(const U8 * pData, unsigned NumBytes) {
    USB_MEMCPY(_acData, pData, NumBytes);
    _acData[NumBytes] = 0;
    printf(_acData);
    return 0;
}

/*****
 *
 *      _OnReset
 *
 */
static void _OnReset(void) {

}

static USB_PRINTER_API _PrinterAPI = {
    _GetDeviceIdString,
    _OnDataReceived,
    _GetHasNoError,
    _GetIsSelected,
    _GetIsPaperEmpty,
    _OnReset
};

/*****
 *
 *      Public code
 *
 */

/*****
 *
 *      USB_GetVendorId
 *
 *      Function description
 *      Returns vendor Id
 */
U16 USB_GetVendorId(void) {
    return 0x8765;
}

/*****
 *
 *      USB_GetProductId
 *
 *      Function description
 *      Returns product Id
 */
U16 USB_GetProductId(void) {
    return 0x2114;    // Should be unique for this sample
}

/*****
 *
 *      USB_GetVendorName
 *
 *      Function description
 *      Returns vendor name
 */
const char * USB_GetVendorName(void) {
    return "Vendor";
}

```

```

/*****
*
*      USB_GetProductName
*
*      Function description
*      Returns product name
*/
const char * USB_GetProductName(void) {
    return "Printer";
}

/*****
*
*      USB_GetSerialNumber
*
*      Function description
*      Returns serial number
*/
const char * USB_GetSerialNumber(void) {
    return "12345678901234567890";
}

/*****
*
*      MainTask
*
*      Function description
*      USB handling task.
*      Modify to implement the desired protocol
*/
void MainTask(void);
void MainTask(void) {
    USB_Init();
    USB_PRINTER_Init(&_PrinterAPI);
    USB_Start();
    //
    // Loop: Receive data byte by byte, send back (data + 1)
    //
    while (1) {
        //
        // Wait for configuration
        //
        while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
            BSP_ToggleLED(0);
            USB_OS_Delay(50);
        }
        USB_PRINTER_Task();
    }
}
/***** end of file *****/

```

9.3 Target API

This chapter describes the functions and data structures that can be used with the target application.

9.3.1 Interface function list

Function	Description
API functions	
<code>USB_PRINTER_Init()</code>	Initializes the printer class.
<code>USB_PRINTER_Task()</code>	Processes the request from USB Host.
Data structures	
<code>USB_PRINTER_API</code>	List of callback functions the library should invoke when processing a request from the USB Host.

Table 9.1: USB-Printer interface API

9.3.2 API functions

9.3.2.1 USB_PRINTER_Init()

Description

Initializes the printer class.

Prototype

```
void USB_PRINTER_Init(USB_PRINTER_API * pAPI);;
```

Parameter	Description
pAPI	Pointer to an API table that contains all callback functions that are necessary for handling the functionality of a printer.

Table 9.2: USB_PRINTER_Init() parameter list

Additional information

After the initialization of general emUSB, this is the first function that needs to be called when the printer class is used with emUSB.

9.3.2.2 USB_PRINTER_Task()

Description

Processes the request received from the USB Host.

Prototype

```
void USB_PRINTER_Task(void);
```

Additional information

This function blocks as long as the USB device is connected to USB host. It handles the requests by calling the function registered in the call to [USB_PRINTER_Init\(\)](#).

9.3.3 Data structures

9.3.3.1 USB_PRINTER_API

Description

Initialization structure that is needed when adding a printer interface to emUSB. It holds pointer to callback functions the interface invokes when it processes request from USB host.

Prototype

```
typedef struct {
    const char * (*pfGetDeviceIdString) (void);
    int          (*pfOnDataReceived) (const U8 * pData, unsigned NumBytes);
    U8           (*pfGetHasNoError) (void);
    U8           (*pfGetIsSelected) (void);
    U8           (*pfGetIsPaperEmpty) (void);
    void         (*pfOnReset) (void);
} USB_PRINTER_API;
```

Member	Description
<code>pfGetDeviceIdString</code>	The library calls this function when the USB host requests the printer's identification string. This string should confirm to the IEEE1284 Device Id Syntax: Example: "CLASS:PRINTER;MODEL:HP LaserJet 6MP;MANUFACTURER:Hewlett-Packard;DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;"
<code>pfOnDataReceived</code>	This function is called when data arrives from USB host.
<code>pfGetHasNoError</code>	This function should return a non-zero value if the printer has no error.
<code>pfGetIsSelected</code>	This function should return a non-zero value if the printer is selected.
<code>pfGetIsPaperEmpty</code>	This function should return a non-zero value if the printer is out of paper.
<code>pfOnReset</code>	The library calls this function if the USB host sends a soft reset command.

Table 9.3: USB_PRINTER_API elements

Chapter 10

Combining different USB components (Multi-Interface)

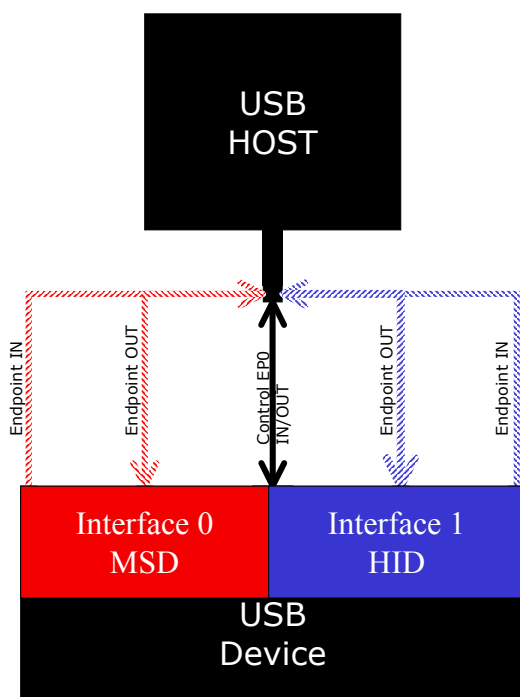
In some cases, it is necessary to combine different USB components in one device. This chapter will show how to do this and which steps are necessary.

10.1 Overview

The USB specification allows to implement more than one component (function) in a single device. This is done by combining two or more components. These devices will be recognized by the USB host as composite device and each component will be recognized as an independent device.

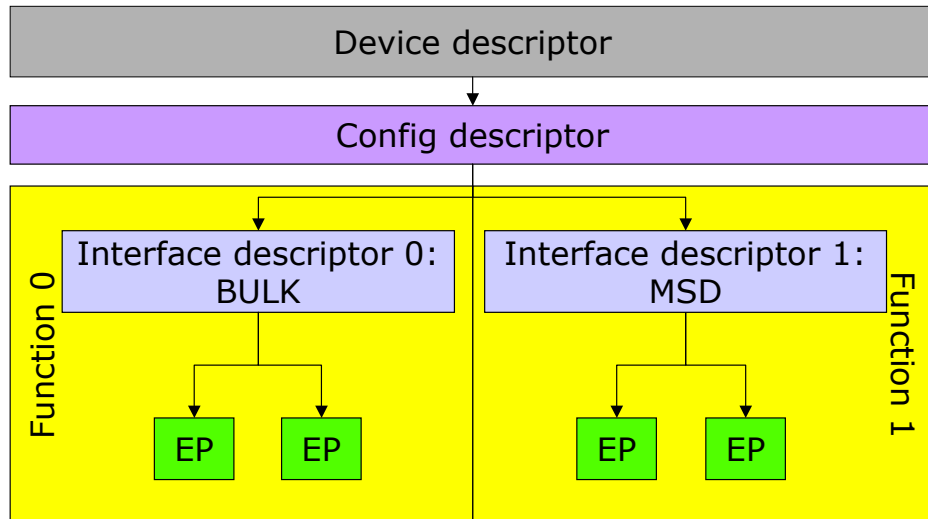
One device for example, a data logger can have two components:

This device can show log data files that were stored on a NAND flash through the MSD component. The configuration of the data logger can be changed by using a BULK component, CDC component or even HID component.



10.1.1 Single interface device classes

Components can be combined because most USB device classes are based on one interface. This means that those components describe themselves on interface descriptor level and thus makes it easy to combine different or even same device classes into one device. Such devices classes are MSD, HID and generic bulk class.

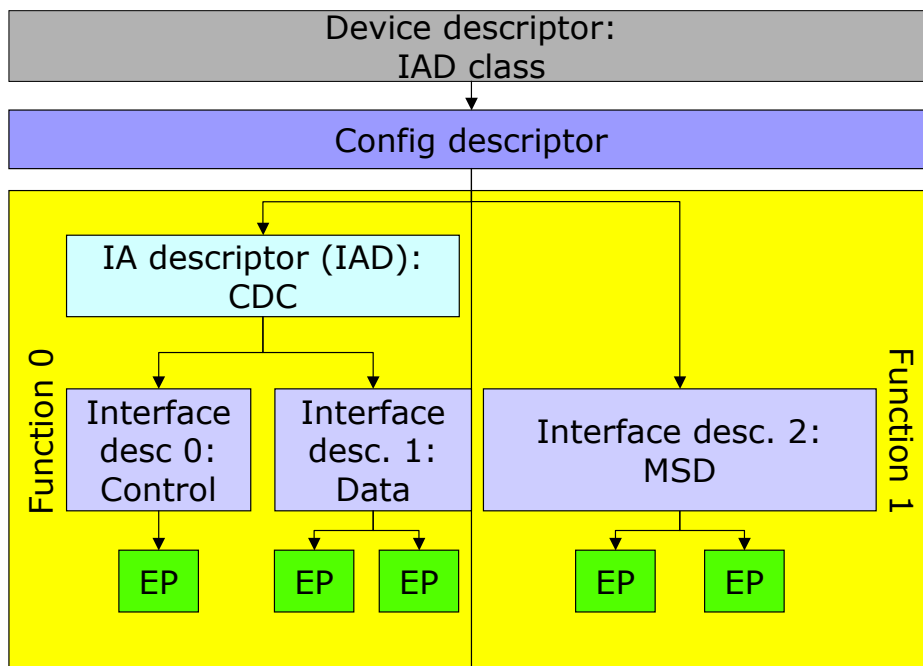


10.1.2 Multiple interface device classes

In contrast to the single interfaces classes there are classes with multiple interfaces such as CDC and AUDIO or VIDEO class. These classes define their class identifier in the device descriptor. All interface descriptors are recognized as part of the component that is defined in the device descriptor. This prevents the combination of multiple interface device classes (for example, CDC) any other component.

10.1.3 IAD class

To remove this limitation the USB organization has defined a new descriptor type that allows the combination of single interface device classes with multiple interface device classes. This descriptor is called Interface Association Descriptor (IAD). It decouples the multi interface class from other interfaces.



Since IAD is an extension to the original USB specification, it is not supported by all hosts, especially older host software. If IAD is not supported, the device may not be enumerated correctly.

Supported HOST

At the time of writing, IAD is supported by:

- Windows XP + Service pack 2
- Windows 2003 + Service pack 1
- Windows Vista
- Linux Kernel 2.6.22 and higher
- Windows 7

10.2 Configuration

In general, no configuration is required. By default, emUSB supports up to 4 interfaces. If more interfaces are needed the following macro needs to be modified:

Type	Macro	Default	Description
Numeric	USB_MAX_NUM_IF	4	Defines the maximum number of interfaces emUSB should handle.
Numeric	USB_MAX_NUM_IAD	1	Defines the maximum number of Interface Association Descriptors emUSB should handle.

10.3 How to combine

Combining different single interface emUSB components (Bulk, HID, MSD) is an easy step, all that needs to be done is calling appropriate `USB_xxx_Add()` function. For adding the CDC component additional steps needs to be done. For detailed information, refer to *emUSB component specific modification* on page 256 and check the following sample.

Requirements

- RTOS, every component requires a separate task.
- Sufficient endpoints all used device classes. Make sure that your USB device controller has enough endpoints available to handle all the interfaces that should be integrated.

Sample application

The following sample application uses embOS as operating system. This listing is excerpt from `USB_CompositeDevice_MSD_CDC.c`.

```
#include "RTOS.H"
#include "USB.h"
#include "USB_MSD.h"
#include "USB_CDC.h"

static OS_STACKPTR int Stack0[512]; /* Task stacks for MSD task */
static OS_STACKPTR int Stack1[512]; /* Task stacks for CDC task */
static OS_TASK TCB0; /* Task-control-block for MSD task*/
static OS_TASK TCB1; /* Task-control-block for CDC task*/

/*****
 *
 *      _CDCTask
 */
static void _CDCTask(void) {
    while (1) {
        char ac[64];
        int NumBytesReceived;
        //
        // Wait for configuration
        //
        while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
            USB_OS_Delay(50);
        }
        NumBytesReceived = USB_CDC_Receive(&ac[0], sizeof(ac));
        if (NumBytesReceived > 0) {
            USB_CDC_Write(&ac[0], NumBytesReceived);
        }
    }
}

/*****
 *
 *      _MSDTask
 */
static void _MSDTask(void) {
    while (1) {
        while ((USB_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
            USB_OS_Delay(50);
        }
        USB_MSD_Task();
    }
}
```

```

/*****
*
*      _AddCDC
*/
static void _AddCDC(void) {
    static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
    USB_CDC_INIT_DATA    InitData;

    InitData.EPIn  = USB_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
    InitData.EPOut = USB_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_BULK, 0,
                               _abOutBuffer, USB_MAX_PACKET_SIZE);
    InitData.EPInt = USB_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_INT,  8,  NULL, 0);
    USB_CDC_Add(&InitData);
}

/*****
*
*      _AddMSD
*/
static void _AddMSD(void) {
    static U8 _abOutBuffer[USB_MAX_PACKET_SIZE];
    USB_MSD_INIT_DATA    InitData;
    USB_MSD_INST_DATA    InstData;

    InitData.EPIn  = USB_AddEP(1, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
                               NULL, 0);
    InitData.EPOut = USB_AddEP(0, USB_TRANSFER_TYPE_BULK, USB_MAX_PACKET_SIZE,
                               _abOutBuffer, USB_MAX_PACKET_SIZE);

    USB_MSD_Add(&InitData);
    memset(&InstData, 0, sizeof(InstData));
    InstData.pAPI          = &USB_MSD_StorageByName;
    InstData.DriverData.pStart = "";
    USB_MSD_AddUnit(&InstData);
}

/*****
*
*      main
*/
void main(void) {
    OS_IncDI();
    OS_InitKern();
    OS_InitHW();
    USB_Init();
    USB_EnableIAD();
    _AddMSD();
    _AddCDC();
    USB_Start();
    OS_CREATETASK(&TCB0, "MSDTask", _MSDTask, 100, Stack0);
    OS_CREATETASK(&TCB1, "CDCTask", _CDCTask,  50, Stack1);
    OS_Start()
}

```

10.4 emUSB component specific modification

There are different steps to do for each emUSB component. The next section shows what needs to be done on both sides: Device and host-side.

10.4.1 BULK communication component

10.4.1.1 Device side

No modification on device side needs to be made.

10.4.1.2 Host side

Windows will recognize the device as a composite device. It will load the drivers for each interface.

In order to recognize the bulk interface in the composite device, the `.inf` file of the device needs to be modified.

Windows will extend the device identification string with the interface number. This has to be added to the device identification string in the `.inf` file.

The provided `.inf` file:

```
;
; Generic USBBulk driver setup information file
; Copyright (c) 2006-2008 by SEGGER Microcontroller GmbH & Co. KG
;
; This file supports:
;   Windows 2000
;   Windows XP
;   Windows Server 2003 x86
;   Windows Vista x86
;   Windows Server 2008 x86
;
[Version]
Signature="$Windows NT$"
Provider=%MfgName%
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
DriverVer=03/19/2008,2.6.6.0
CatalogFile=USBBulk.cat

[Manufacturer]
%MfgName%=DeviceList

[DeviceList]
%USB\VID_8765&PID_1234.DeviceDesc%=USBBulkInstall, USB\VID_8765&PID_1234&Mi_xx

[USBBulkInstall.ntx86]
CopyFiles=USBBulkCopyFiles

[USBBulkInstall.ntx86.Services]
Addservice = usbbulk, 0x00000002, USBBulkAddService, USBBulkEventLog

[USBBulkAddService]
DisplayName     = %USBBulk.SvcDesc%
ServiceType     = 1                ; SERVICE_KERNEL_DRIVER
StartType       = 3                ; SERVICE_DEMAND_START
ErrorControl    = 1                ; SERVICE_ERROR_NORMAL
ServiceBinary   = %10%\System32\Drivers\USBBulk.sys

[USBBulkEventLog]
AddReg=USBBulkEventLogAddReg

[USBBulkEventLogAddReg]
HKR,,EventMessageFile,%REG_EXPAND_SZ%, "%SystemRoot%\System32\IoLogMsg.dll;%%System
Root%\System32\drivers\USBBulk.sys"
HKR,,TypesSupported, %REG_DWORD%,7

[USBBulkCopyFiles]
USBBulk.sys

[DestinationDirs]
DefaultDestDir = 10,System32\Drivers
USBBulkCopyFiles = 10,System32\Drivers
```

```

[SourceDisksNames.x86]
1=%USBBulk.DiskName%, ,

[SourceDisksFiles.x86]
USBBulk.sys = 1

;-----;

[Strings]
MfgName="Segger"
USB\VID_8765&PID_1234.DeviceDesc="USB Bulk driver"
USBBulk.SvcDesc="USBBulk driver"
USBBulk.DiskName="USBBulk Installation Disk"

; Non-Localizable Strings, DO NOT MODIFY!
REG_SZ          = 0x00000000
REG_MULTI_SZ    = 0x00010000
REG_EXPAND_SZ   = 0x00020000
REG_BINARY      = 0x00000001
REG_DWORD       = 0x00010001

; *** EOF ***

```

Please add the red colored text to your .inf file and change xx with the interface number of the bulk component.

The interface number is a zero based index and is assigned by the emUSB stack when calling `USB_BULK_Add()` function. If you have called `USB_BULK_Add()` prior any other `USB_XXX_Add()` functions then the interface number will be 00.

Please note that when `USB_CDC_Add()` is called prior `USB_BULK_Add()`, the interface number for the BULK component will be 02 since the CDC component uses two interfaces (in the example, 00 and 01).

10.4.2 MSD component

10.4.2.1 Device side

No modification on device side needs to be made.

10.4.2.2 Host side

No modification on host side needs to be made.

10.4.3 CDC component

10.4.3.1 Device side

In order to combine the CDC component with other components, the function `USB_EnableIAD()` needs to be called, otherwise the device will not enumerate correctly. Refer to section *How to combine* on page 254 and check the listing of the sample application.

10.4.3.2 Host side

Due to a limitation of the internal CDC serial driver of Windows, a composite device with CDC component and an other device component(s) is properly recognized by Windows XP SP3 and Windows Vista and above. Linux kernel supports IAD with version 2.6.22.

For Windows Operation system the `.inf` file needs to be modified.

As in the Bulk communication component, Windows will extends the device identification strings. Therefore the device identification string has to be modified.

The provided `.inf` file:

```
;
; Device installation file for
; USB 2 COM port emulation
;
;
;
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
LayoutFile=layout.inf
DriverVer=03/26/2007,6.0.2600.1
CatalogFile=usbser.cat

[Manufacturer]
%MFGNAME%=CDCDevice,NT,NTamd64

[DestinationDirs]
DefaultDestDir = 12

[CDCDevice.NT]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[CDCDevice.NTamd64]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_0234&Mi_xx
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[DriverInstall.NT]
```

```

Include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=DriverInstall.NT.AddReg

[DriverInstall.NT.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,, "MsPorts.dll,SerialPortPropPageProvider"

[DriverInstall.NT.Services]
AddService=usbser, 0x00000002, DriverServiceInst

[DriverServiceInst]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
MFGNAME = "Manufacturer"
DESCRIPTION = "USB CDC serial port emulation"
SERVICE = "USB CDC serial port emulation"

```

Please add the red colored text to your .inf file and change xx with the interface number of the CDC component.

The interface number is a zero based index and is assigned by the emUSB stack when calling USB_CDC_Add() function.

10.4.4 HID component

10.4.4.1 Device side

No modification on device side needs to be made.

10.4.4.2 Host side

No modification on host device side needs to be made.

Chapter 11

Target OS Interface

This chapter describes the functions of the operating system abstraction layer.

11.1 General information

emUSB includes an OS abstraction layer which should make it possible to use an arbitrary operating system together with emUSB. To adapt emUSB to a new OS one has only to map the functions listed below in section *Interface function list* on page 263 to the native OS functions.

Segger took great care when designing this abstraction layer, to make it easy to understand and to adapt to different operating systems.

11.1.1 Operating system support supplied with this release

In the current version, abstraction layers for embOS and μ C/OS-II are available.

11.2 Interface function list

Routine	Explanation
USB_OS_Delay()	Delays for a given number of ms.
USB_OS_DecRI()	Decrement interrupt disable count and enable interrupts if counter reaches 0.
USB_OS_GetTickCnt()	Returns the current system time in ticks.
USB_OS_IncDI()	Increment interrupt disable count and disable interrupts.
USB_OS_Init()	Initializes OS.
USB_OS_Panic()	Called if fatal error is detected.
USB_OS_Signal()	Wake the task waiting for signal.
USB_OS_Wait()	Block the task until USB_OS_Signal() is called.
USB_OS_WaitTimed()	Block the task until USB_OS_Signal() is called or a time-out occurs.

Table 11.1: Target OS interface function list

11.2.1 USB_OS_Delay()

Description

Delays for a given number of ms.

Prototype

```
void USB_OS_Delay(int ms);
```

Parameter	Description
ms	Number of ms.

Table 11.2: USB_OS_Delay() parameter list

11.2.2 USB_OS_DecRI()

Description

Decrements interrupt disable count and enables interrupts if counter reaches 0.

Prototype

```
void USB_OS_DecRI(void);
```

11.2.3 USB_OS_GetTickCnt()

Description

Returns the current system time in ticks.

Prototype

```
U32 USB_OS_GetTickCnt(void);
```

11.2.4 USB_OS_IncDI()

Description

Increments interrupt disable count and disables interrupts.

Prototype

```
void USB_OS_IncDI(void);
```

11.2.5 USB_OS_Init()

Description

Initializes OS.

Prototype

```
void USB_OS_Init(void);
```

11.2.6 USB_OS_Panic()

Description

Halts emUSB.

Prototype

```
void USB_OS_Panic(unsigned ErrCode);
```

Parameter	Description
ErrCode	Error code.

Table 11.3: USB_OS_Panic() parameter list

Add. information

Errorcode	Explanation
1	USB_ERROR_RX_OVERFLOW
2	USB_ERROR_ILLEGAL_MAX_PACKET_SIZE
3	USB_ERROR_ILLEGAL_EPADDR
4	USB_ERROR_IBUFFER_SIZE_TOO_SMALL
5	USB_ERROR_DRIVER_ERROR
6	USB_ERROR_IAD_DESCRIPTOR_EXCEED
7	USB_ERROR_INVALID_INTERFACE_NO

Table 11.4: USB_OS_Panic(): Errorcodes

11.2.7 USB_OS_Signal()

Description

Wakes the task waiting for signal.

Prototype

```
void USB_OS_Signal(unsigned EPIndex);
```

Parameter	Description
EPIndex	Endpoint index.

Table 11.5: USB_OS_Signal() parameter list

Add. information

This routine is typically called from within an interrupt service routine.

11.2.8 USB_OS_Wait()

Description

Blocks the task until `USB_OS_Signal()` is called.

Prototype

```
void USB_OS_Wait(unsigned EPIndex);
```

Parameter	Description
EPIndex	Endpoint index.

Table 11.6: USB_OS_Wait() parameter list

Add. information

This routine is called from a task.

11.2.9 USB_OS_WaitTimed()

Description

Blocks the task until `USB_OS_Signal()` is called or a time-out occurs.

Prototype

```
int USB_OS_WaitTimed(unsigned EPIndex, unsigned ms);
```

Parameter	Description
<code>EPIndex</code>	Endpoint index.
<code>ms</code>	Time-out time given in ms.

Table 11.7: USB_OS_Wait() parameter list

Return value

== 0: Task was signaled within the given time-out.

== 1: Time-out occurred.

Add. information

`USB_OS_WaitTimed` is also called from a task. This function is used by all available timed routines.

11.3 Example

A configuration to use USB with embOS might look like the sample below. This example is also supplied in the subdirectory `OS\embOS\`.

```

/*****
*
*           SEGGER Microcontroller GmbH & Co. KG
*           Solutions for real time microcontroller applications
*****
*
*           (C) 2003 - 2004   SEGGER Microcontroller GmbH & Co. KG
*
*           www.segger.com     Support: support@segger.com
*
*****
*
*           emUSB stack for embedded applications
*
*****

-----
File      : USB_OS_embOS.c
Purpose  : Kernel abstraction for embOS
          Do not modify to allow easy updates !
-----
END-OF-HEADER -----
*/

#include "USB_Private.h"
#include "RTOS.h"

/*****
*
*           Static data
*
*****
*/
static OS_EVENT _aEvent[USB_NUM_EPS];

/*****
*
*           Public code
*
*****
*/

/*****
*
*           USB_OS_Init
*
*/
void USB_OS_Init(void) {
    unsigned i;

    for (i = 0; i < COUNTOF(_aEvent); i++) {
        OS_EVENT_Create(&_aEvent[i]);
    }
}

/*****
*
*           USB_OS_Delay
*
*****/

```

```

* Function description
*   Delays for a given number of ms.
*/
void USB_OS_Delay(int ms) {
    OS_Delay(ms);
}

/*****
*
*   USB_OS_DecRI
*
* Function description
*   Decrement interrupt disable count and enable interrupts
*   if counter reaches 0
*/
void USB_OS_DecRI(void) {
    OS_DecRI();
}

/*****
*
*   USB_OS_IncDI
*
* Function description
*   Increment interrupt disable count and disable interrupts
*/
void USB_OS_IncDI(void) {
    OS_IncDI();
}

/*****
*
*   USB_OS_Signal
*
* Function description
*   Wake the task waiting for reception
*
* Add. info
*   This routine is typically called from within an interrupt
*   service routine
*/
void USB_OS_Signal(unsigned EPIndex) {
    OS_EVENT_Pulse(&aEvent[EPIndex]);
}

/*****
*
*   USB_OS_Wait
*
* Function description
*   Block the task until USB_OS_Signal is called
* Add. info
*   This routine is called from a task.
*/
void USB_OS_Wait(unsigned EPIndex) {
    OS_EVENT_Wait(&aEvent[EPIndex]);
}

/*****
*
*   USB_OS_Panic

```

```

*
* Function description
*   Called if fatal error is detected.
*
* Add. info
*   Error codes:
*     1   USB_ERROR_RX_OVERFLOW
*     2   USB_ERROR_ILLEGAL_MAX_PACKET_SIZE
*     3   USB_ERROR_ILLEGAL_EPADDR
*     4   USB_ERROR_IBUFFER_SIZE_TOO_SMALL
*/
void USB_OS_Panic(unsigned ErrCode) {
    while (ErrCode);
}

/*****
*
*       USB_OS_GetTickCnt
*
* Function description
*   Returns the current system time in ticks.
*/
U32 USB_OS_GetTickCnt(void) {
    return OS_Time;
}

/***** End of file *****/

```


Chapter 12

Target USB Driver

This chapter describes emUSB hardware interface functions in detail.

12.1 General information

Purpose of the USB hardware interface

emUSB does not contain any hardware dependencies. These are encapsulated through a hardware abstraction layer, which consists of the interface functions described in this chapter. All of these functions for a particular USB controller are typically located in a single file, the USB driver. Drivers for hardware which has already been tested with emUSB are available.

Range of supported USB hardware

The interface has been designed in such a way that it should be possible to use the most common USB device controllers. This includes USB 1.1 (full speed) controllers, USB 2.0 (high speed) controllers, both as external chips and as part of microcontrollers.

12.1.1 Available USB drivers

The following device drivers are available for emUSB:

Driver (Device)	Identifier
ATMEL AV32 UC3x	USB_Driver_Atmel_AT32UC3x
ATMEL AT91CAP9x	USB_Driver_Atmel_CAP9
ATMEL AT91SAM3Uxx	USB_Driver_Atmel_SAM3US
ATMEL AT91RM9200	USB_Driver_Atmel_RM9200
ATMEL AT91SAM7A3	USB_Driver_Atmel_SAM7A3
ATMEL AT91SAM7S64	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7S128	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7S256	USB_Driver_Atmel_SAM7S
ATMEL AT91SAM7SE	USB_Driver_Atmel_SAM7SE
ATMEL AT91SAM7X128	USB_Driver_Atmel_SAM7X
ATMEL AT91SAM7X256	USB_Driver_Atmel_SAM7X
ATMEL AT91SAM9260	USB_Driver_Atmel_SAM9260
ATMEL AT91SAM9261	USB_Driver_Atmel_SAM9261
ATMEL AT91SAM9263	USB_Driver_Atmel_SAM9263
ATMEL AT91SAM9R64	USB_Driver_Atmel_SAMRx64
ATMEL AT91SAM9RL64	USB_Driver_Atmel_SAMRx64
ATMEL AT91SAM9G20	USB_Driver_Atmel_SAM9G20
ATMEL AT91SAM9G45	USB_Driver_Atmel_SAM9G45
ATMEL AT91SAM9XE	USB_Driver_Atmel_SAM9XE
Freescale iMX25x	USB_Driver_Freescale_iMX25x
Freescale iMX28x	USB_Driver_Freescale_iMX28x
Freescale Kinetis K40	USB_Driver_Freescale_K40
Freescale Kinetis K60	USB_Driver_Freescale_K60
Freescale MCF227x	USB_Driver_Freescale_MCF227x
Freescale MCF225x	USB_Driver_Freescale_MCF225x
Freescale MCF51JMx	USB_Driver_Freescale_MCF51JMx
Fujitsu MB9BF50x	USB_Driver_Fujitsu_MB9BF50x
NXP LPC13xx	USB_Driver_NXPLPC13xx
NXP LPC17xx	USB_Driver_NXPLPC17xx
NXP LPC214x	USB_Driver_NXPLPC214x
NXP LPC23xx	USB_Driver_NXPLPC23xx
NXP LPC24xx	USB_Driver_NXPLPC24xx
NXP LPC313x	USB_Driver_NXPLPC313x

Table 12.1: List of included USB device drivers

Driver (Device)	Identifier
NXP LPC318x	USB_Driver_NXPLPC318x
NXP (formerly Sharp) LH79524/5	USB_Driver_SharpLH79524
NXP (formerly Sharp) LH7A40x	USB_Driver_SharpLH7A40x
OKI 69Q62	USB_Driver_OKI69Q62
Renesas H8S2472	USB_Driver_H8S2472
Renesas H8SX1668R	USB_Driver_H8SX1668R
Renesas RX62N	USB_Driver_RX62N
Renesas SH7203	USB_Driver_SH7203
Renesas SH7216	USB_Driver_SH7216
Renesas SH7286	USB_Driver_SH7286
Renesas (NEC) 78K0R-KE3L	USB_Driver_NEC_78F102x
Renesas (NEC) μ PD720150	USB_Driver_NEC_uPD720150
Renesas (NEC) V850ESJG3H	USB_Driver_NEC_70F376x
ST STM32	USB_Driver_STSTM32
ST STM32F105/107	USB_Driver_STSTM32F107
ST STR71x	USB_Driver_STSTR71x
ST STR750	USB_Driver_STSTR750
ST STR912	USB_Driver_STSTR91x
Toshiba TMPA900	USB_Driver_TMPA900
Toshiba TMPA910	USB_Driver_TMPA910
Texas Instruments MSP430X5529	USB_Driver_TI_MSP430
Texas Instruments (Luminary) LM3S9B9x	USB_Driver_TI_LM3S9B9x

Table 12.1: List of included USB device drivers

12.2 Adding a driver to emUSB

You have to specify the USB device driver which should be used with emUSB. To specify the driver `USB_X_AddDriver()` is called from `USB_Init()`. This function should be used to add a USB driver to your project.

`USB_Init()` initializes the internal of the USB stack and is always the first function which that USB application has to call. `USB_X_HWAttach()` should be used to perform hardware-specific actions which are not part of the USB controller logic (for example, enabling the peripheral clock for USB port).

This function is called from every device driver, but can be empty if your hardware does not need to perform such actions. Modify `USB_X_AddDriver()` and if required, `USB_X_HWAttach()`. In `USB_X_AddDriver()`, `USB_AddDriver()` should be called with the identifier of the driver which is compatible to your hardware as parameter. Refer to the section *Available USB drivers* on page 278 for a list of all supported devices and their valid identifiers.

12.2.1 USB_X_HWAttach()

Description

Should be used to perform hardware specific actions which are not part of the USB controller logic.

Prototype

```
void USB_X_HWAttach(void)
```

Additional Information

This function can be empty, if no hardware-specific actions are required.

Example

```
/* Example excerpt from USB_Config_SAM7A3.c */#

#define _AT91C_PIOA_BASE (0xFFFFF400)
#define _AT91C_PIOB_BASE (0xFFFFF600)
#define _AT91C_PMC_BASE (0xFFFFFC00)
#define _PIO_PER_OFFS (0x00)
#define _PIO_OER_OFFS (0x10)
#define _PIO_CODR_OFFS (0x34) /* Clear output data register */
#define _PMC (*(volatile unsigned int*) _AT91C_PMC_BASE)
#define _USB_ID (_PIOB_ID)
#define _USB_OER (*(volatile unsigned int*) (_AT91C_PIOB_BASE + _PIO_OER_OFFS))
#define _USB_CODR (*(volatile unsigned int*) (_AT91C_PIOB_BASE + _PIO_CODR_OFFS))
#define _USB_DP_PUP_BIT (1)

void USB_X_HWAttach(void) {
    _PMC = (1 << _USB_ID); /* Enable peripheral clock for USB-Port */
    _USB_OER = (1 << _USB_DP_PUP_BIT); /* set USB_DP_PUP to output */
    _USB_CODR = (1 << _USB_DP_PUP_BIT); /* set _USB_DP_PUP_BIT to low state */
}
```

12.2.2 USB_X_AddDriver()

Description

Adds a USB hardware driver to the USB stack.

Prototype

```
void USB_X_AddDriver(void)
```

Additional information

This function is always called from `USB_Init()`.

Example

```
/* Example excerpt from USB_Config_SAM7A3.c */  
  
void USB_X_AddDriver(void) {  
    USB_AddDriver(&USB_Driver_Atme1SAM7A3);  
}
```

12.3 Interrupt handling

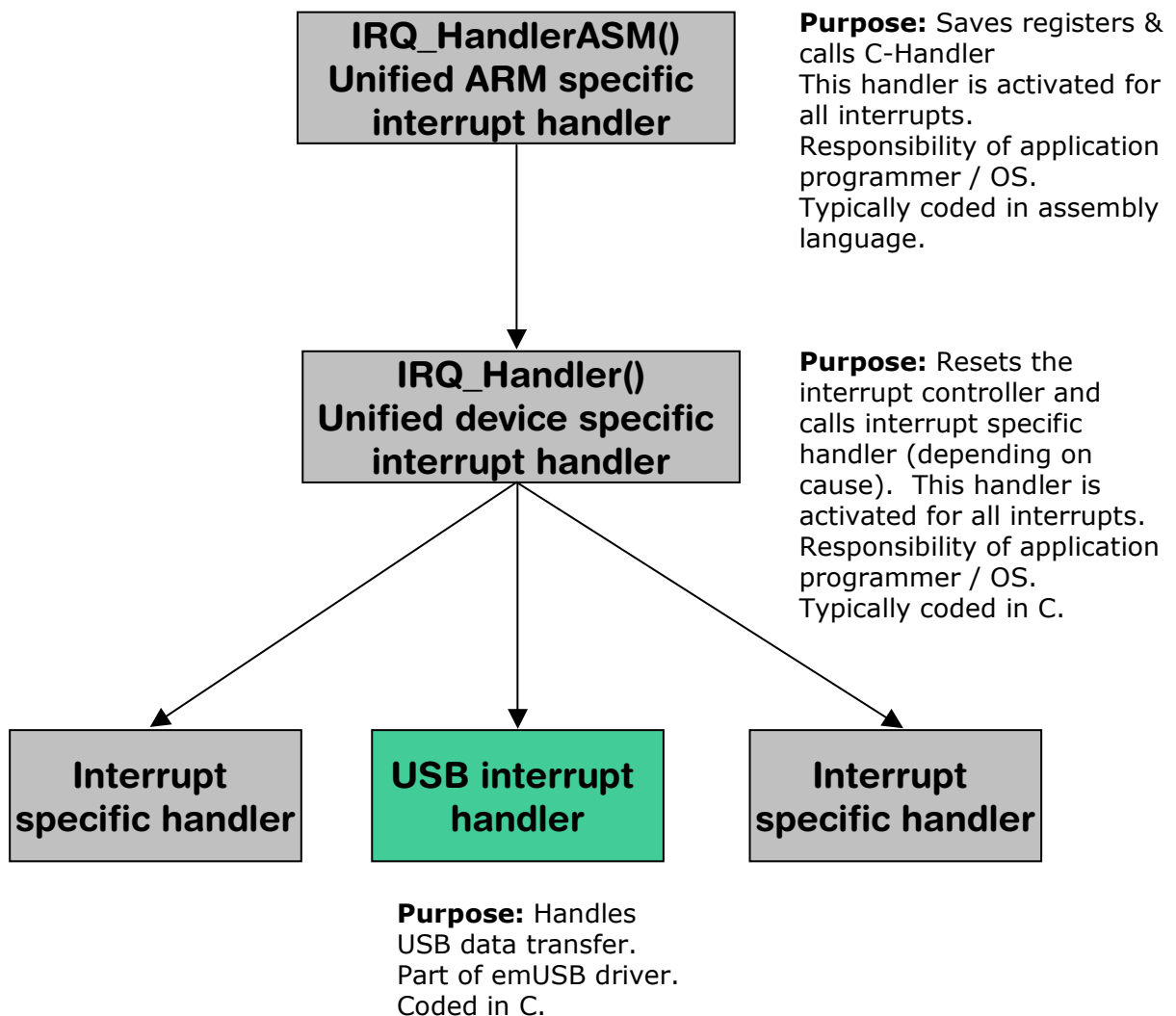
emUSB is interrupt driven and optimized to be used with a real-time operating system. If you use embOS in combination with emUSB, you can skip the following sections.

If you are not using embOS, you have to be familiar with how interrupts are handled on your target system. This includes knowledge about how the CPU handles interrupts, how and which registers are saved, the interrupt vector table, how the interrupt controller works and how it is reset.

12.3.1 ARM7 / ARM9 based cores

ARM7 and ARM9 cores will jump to IRQ vector address 0×18 , where a jump to an ARM specific IRQ handler should be located. This ARM specific IRQ handler calls a device specific interrupt handler which handles the interrupt controller.

The ARM specific interrupt handler is typically coded in assembly language. It has to ensure that no context information will be lost if an interrupt occurs. The environment of the interrupted function has to be restored after processing the interrupt. The environment of the interrupted function includes the value of the processor registers and the processor status register. The ARM specific interrupt handler calls a high-level interrupt handler which manages the call of the interrupt source specific service routine.



12.3.1.1 ARM specific IRQ handler

The ARM specific interrupt handler saves the context of the function which is interrupted, calls the high-level interrupt handler and restores the context. Sample implementations of the high-level handler are supplied in the following device specific sections.

Sample implementation interrupt handler

```

EXTERN  IRQ_Handler

IRQ_HandlerASM:
;
; Save temp. registers
;
    stmdb    SP!, {R0-R3,R12,LR}          ; push
;
; push SPSR (req. if we allow nested interrupts)
;
    mrs     R0, SPSR                      ; load SPSR
    stmdb  SP!, {R0}                      ; push SPSR_irq on IRQ stack
;
; Call "C" interrupt handler
;
    ldr     R0, =IRQ_Handler
    mov    LR, PC
    bx     R0
;
; pop SPSR
;
    ldmia  SP!, {R1}                      ; pop SPSR_irq from IRQ stack
    msr    SPSR_cxfs, R1
;
; Restore temp registers
;
    ldmia  SP!, {R0-R3,R12,LR}          ; pop
    subs   PC, LR, #4                    ; RETI

```

12.3.1.2 Device specifics ATMEL AT91CAP9x

The interrupt handler needs to read the address of the interrupt source specific handler function.

Sample implementation interrupt handler

```
#define _AIC_BASE_ADDR      (0xfffff000UL)
#define _AIC_IVR           (*(volatile unsigned int*)(_AIC_BASE_ADDR + 0x100))
#define _AIC_EOICR        (*(volatile unsigned int*)(_AIC_BASE_ADDR + 0x130))

typedef void      ISR_HANDLER(void);

void IRQ_Handler(void) {
    ISR_HANDLER* pISR;
    pISR = (ISR_HANDLER*) _AIC_IVR;          // Read interrupt vector to release
                                              // NIRQ to CPU core
    pISR();                                  // Call interrupt service routine
    _AIC_EOICR = 0;                          // Reset interrupt controller => Restore
                                              // previous priority
}
```

Additional information

This example can also be used with an ATMEL AT91RM9200, AT91SAM7A3, AT91SAM7S64, AT91SAM7S128, AT91SAM7S256, AT91SAM7SE, AT91SAM7X64, AT91SAM7X128, AT91SAM7X256, and AT91SAM9261.

12.3.1.3 Device specifics ATMEL AT91RM9200

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.4 Device specifics ATMEL AT91SAM7A3

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.5 Device specifics ATMEL AT91SAM7S64, AT91SAM7S128, AT91SAM7S256

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.6 Device specifics ATMEL AT91SAM7X64, AT91SAM7X128, AT91SAM7X256

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.7 Device specifics ATMEL AT91SAM7SE

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.8 Device specifics ATMEL AT91SAM9260

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.9 Device specifics ATMEL AT91SAM9261

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.10 Device specifics ATMEL AT91SAM9263

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.11 Device specifics ATMEL AT91SAMRL64, AT91SAMR64

For an example implementation of an interrupt handler function refer to *Device specifics ATMEL AT91CAP9x* on page 285.

12.3.1.12 Device specifics NXP LPC214x

The interrupt handler needs to read the address of the interrupt source specific handler function.

Sample implementation interrupt handler

```
#define _VIC_BASE_ADDR      (0xFFFFF000)
#define _VIC_VECTORADDR    *(volatile unsigned int*)(_VIC_BASE_ADDR + 0x0030)

typedef void      ISR_HANDLER(void);

void IRQ_Handler(void) {
    ISR_HANDLER* pISR;
    pISR = (ISR_HANDLER*) _VIC_VECTORADDR;      // Get current interrupt handler
    pISR();                                     // Call interrupt service routine
    _VIC_VECTORADDR = 0;                       // Clear current interrupt pending
                                                // condition, reset VIC
}
```

12.3.1.13 Device specifics NXP LPC23xx

For an example implementation of an interrupt handler function refer to *Device specifics NXP LPC214x* on page 287.

12.3.1.14 Device specifics NXP (formerly Sharp) LH79524/5

For an example implementation of an interrupt handler function, please contact SEGGER, www.segger.com.

12.3.1.15 Device specifics OKI 69Q62

For an example implementation of an interrupt handler function, please contact SEGGER, www.segger.com.

12.3.1.16 Device specifics ST STR71x

For an example implementation of an interrupt handler function, please contact SEGGER, www.segger.com.

12.3.1.17 Device specifics ST STR750

For an example implementation of an interrupt handler function, please contact SEGGER, www.segger.com.

12.3.1.18 Device specifics ST STR750

For an example implementation of an interrupt handler function, please contact SEGGER, www.segger.com.

12.4 Writing your own driver

This section is only relevant if you plan to develop a driver for an unsupported device. Refer to *Available USB drivers* on page 278 for a list of currently supported devices.

Access to the USB hardware is realized through an API-function table. The structure `USB_HW_DRIVER` is declared in `USB\USB.h`.

12.4.1 Structure `USB_HW_DRIVER`

Description

Structure that contains callback function which manage the hardware access.

Prototype

```
typedef struct USB_HW_DRIVER {
    void      (*pfInit)                (void);
    U8        (*pfAllocEP)             (U8 InDir, U8 TransferType);
    void      (*pfUpdateEP)            (EP_STAT * pEPStat);
    void      (*pfEnable)               (void);
    void      (*pfAttach)               (void);
    unsigned  (*pfGetMaxPacketSize)    (U8 EPIndex);
    int       (*pfIsInHighSpeedMode)   (void);
    void      (*pfSetAddress)           (U8 Addr);
    void      (*pfSetClrStalleP)       (U8 EPIndex, int OnOff);
    void      (*pfStalleP0)             (void);
    void      (*pfDisableRxInterruptEP) (U8 EpOut);
    void      (*pfEnableRxInterruptEP) (U8 EpOut);
    void      (*pfStartTx)              (U8 EPIndex);
    void      (*pfSendEP)              (U8 EPIndex, const U8 * p,
                                       unsigned NumBytes);

    void      (*pfDisableTx)           (U8 EPIndex);
    void      (*pfResetEP)             (U8 EPIndex);
    int       (*pfControl)              (U8 Cmd, void * p);
} USB_HW_DRIVER;
```

Member	Description
USB initialization functions	
<code>(*pfInit)()</code>	Initializes the USB controller.
General USB functions	
<code>(*pfAttach)()</code>	Indicates device attachment.
<code>(*pfEnable)()</code>	Enables endpoint.
<code>(*pfControl)()</code>	Used to support additional driver functionality. This function is optional.
<code>(*pfSetAddress)()</code>	Notifies the USB controller of the new address assigned by the host for it.
General endpoints functions	
<code>(*pfAllocEP)()</code>	Allocates an endpoint to be used with emUSB.
<code>(*pfGetMaxPacketSize)()</code>	Returns the maximum packet size of an endpoint.
<code>(*pfSetClrStalleP)()</code>	Set or cleats the stall condition of the endpoint.
<code>(*pfUpdateEP)()</code>	Configures the USB controller's endpoint.
<code>(*pfResetEP)()</code>	Resets an endpoint including resetting the data toggle of the endpoint.
Endpoint 0 (Control endpoint) related functions	

Table 12.2: List of callback functions of `USB_HW_DRIVER`

Member	Description
<code>(*pfStallEP0)()</code>	Stalls endpoint 0.
OUT-endpoints functions	
<code>(*pfDisableRxInterruptEP)()</code>	Disables OUT-endpoint interrupt.
<code>(*pfEnableRxInterruptEP)()</code>	Enables OUT-endpoint interrupt.
IN-endpoints functions	
<code>(*pfStartTx)()</code>	Disables IN endpoint transfers.
<code>(*pfSendEP)()</code>	Sends data on the given IN-endpoint.
<code>(*pfDisableTx)()</code>	Starts data transfer on the given IN-endpoint.

Table 12.2: List of callback functions of USB_HW_DRIVER

12.4.2 USB initialization functions

12.4.2.1 (*pfInit())

Description

Performs any necessary initializations on the USB controller.

Prototype

```
void (*pfInit)(void);
```

Additional information

The initializations performed in this routine should include what is needed to prepare the device for enumeration. Such initializations might include setting up endpoint 0 and enabling interrupts. It sets default values for EP0 and enables the various interrupts needed for USB operations.

12.4.3 General USB functions

12.4.3.1 (*pfAttach)()

Description

For USB controllers that have a USB Attach/Detach register (such as the OKI ML69Q6203), this routine sets the register to indicate that the device is attached.

Prototype

```
void (*pfAttach)(void);
```

12.4.3.2 (*pfEnable)()

Description

This function is used for enabling the USB controller after it was initialized.

Prototype

```
void (*pfEnable)(void);
```

Additional information

For most USB controllers this function can be empty. This function is only necessary for USB devices that reset their configuration data after an USB-RESET.

12.4.3.3 (*pfControl)()

Description

This function is used to support additional driver functionality. This function is optional.

Prototype

```
int (*pfControl)(U8 Cmd, void * p);
```

Parameter	Description
Cmd	Command that should be executed.
p	Pointer to data, necessary for the command.

Table 12.3: (*pfControl)() parameter list

Return value

- 0 - Command operation was successful.
- 1 - Command operation was not successful.
- 1 - Command was unknown.

Additional information

This control function is only called when available. This function will check or changes state of a device driver. Currently the following commands are available:

Command	Description
0	USB_DRIVER_CMD_SET_CONFIGURATION
1	USB_DRIVER_CMD_GET_TX_BEHAVIOR

Table 12.4: (*pfControl): Commands

12.4.3.4 (*pfSetAddress)()

Description

This function is used for notifying the USB controller of the new address that the host has assigned to it during enumeration.

Prototype

```
void (*pfSetAddress)(U8 Addr);
```

Parameter	Description
Addr	New address assigned by the USB host.

Table 12.5: (*pfSetAddress)() parameter list

Additional information

If the USB controller does not automatically send a 0-byte acknowledgment in the status stage of the control transfer phase, make sure to set a state variable to [Addr](#) and defer setting the controller's Address register until after the status stage. This is necessary because the host sends the token packet for the status stage to the default address (0x00), which means the device must still be using this address when the packet is sent.

12.4.4 General endpoint functions

12.4.4.1 (*pfAllocEP)()

Description

Allocates a physical endpoint to be used with emUSB.

Prototype

```
U8 (*pfAllocEP)(U8 InDir, U8 TransferType);
```

Parameter	Description
<code>InDir</code>	Indicates the direction of the endpoint. 0 indicates an OUT-endpoint. 1 indicates an IN-endpoint.
<code>TransferType</code>	Specifies the transfer type for the desired endpoint. The following transfer types are available: USB_TRANSFER_TYPE_BULK USB_TRANSFER_TYPE_ISO USB_TRANSFER_TYPE_INT

Table 12.6: (*pfAllocEP)() parameter list

Return value

Index number of the logical endpoint. Allowed values are 1..15.

Additional information

This function is typically called after stack initialization, in order to have the right endpoint settings for building the descriptors correctly.

It is the responsibility of the driver engineer to give a valid logical endpoint number. If there is no valid endpoint for the desired configuration available, 0 should be returned.

12.4.4.2 (*pfGetMaxPacketSize)()

Description

Returns the maximum packet size of an endpoint.

Prototype

```
unsigned (*pfGetMaxPacketSize)(U8 EPIndex);
```

Parameter	Description
<code>EPIndex</code>	Endpoint index.

Table 12.7: (*pfGetMaxPacketSize)() parameter list

Return value

The maximum packet size in bytes.

12.4.4.3 (*pfSetClrStallEP())

Description

Sets or clears the stall condition of an endpoint.

Prototype

```
void (*pfSetClrStallEP)(U8 EPIndex, int OnOff);
```

Parameter	Description
EPIndex	Endpoint that should be stalled.
OnOff	Specifies if the stall condition should be set or cleared. Whereas: 0 - Clears the stall condition. 1 - Set the stall condition.

Table 12.8: (*pfSetClrStallEP()) parameter list

Additional information

Typically, this function is called whenever a protocol/transfer error occurs.

12.4.4.4 (*pfUpdateEP())

Description

Configures the USB controller's endpoint.

Prototype

```
void (*pfUpdateEP)(EP_STAT * pEPStat);
```

Parameter	Description
pEPStat	Pointer to EP_STAT structure that holds the information for the endpoint.

Table 12.9: (*pfUpdateEP()) parameter list

Additional information

EP_STAT is defined as follows:

```
typedef struct {
    U16      NumAvailBuffers;
    U16      MaxPacketSize;
    U16      Interval;
    U8       EPType;
    BUFFER   Buffer;
    U8       * pData;
    volatile U32 NumBytesRem;
    U8       EPAddr; // b[6:0]: EPAddr b7: Direction, 1: Device to Host (IN)
    U8       Send0PacketIfRequired;
} EP_STAT;
```

Before a hardware attach is done, this function is called to configure the desired endpoints, so that the additional endpoints are ready for use after the enumeration phase.

12.4.4.5 (*pfResetEP)()

Description

Resets an endpoint including resetting the data toggle of the endpoint.

Prototype

```
void (*pfResetEP)(U8 EPIndex);
```

Parameter	Description
EPIndex	Endpoint that should be reset.

Table 12.10: (*pfResetEP)() parameter list

Additional information

Resets the endpoint which includes setting data toggle to DATA0.

It is useful after removing a HALT condition on a BULK endpoint.

Refer to Chapter 5.8.5 in the USB Serial Bus Specification, Rev.2.0.

Note: Configuration of the endpoint needs to be unchanged. If the USB controller loses the EP configuration the `pfUpdateEP` of the driver should be called.

12.4.5 Endpoint 0 (control endpoint) related functions

12.4.5.1 (*pfStalleP0)()

Description

This function is used for stalling endpoint 0 (by setting the appropriate bit in a control register).

Prototype

```
void (*pfStalleP0)(void);
```

12.4.6 OUT-endpoint functions

12.4.6.1 (*pfDisableRxInterruptEP)()

Description

Disables the OUT-endpoint interrupt.

Prototype

```
void (*pfDisableRxInterruptEP)(U8 EPIndex);
```

Parameter	Description
EPIndex	OUT-endpoint whose interrupt needs to be disabled.

Table 12.11: (*pfDisableRXInterruptEP)() parameter list

12.4.6.2 (*pfEnableRxInterruptEP)()

Description

Disables the OUT-endpoint interrupt.

Prototype

```
void (*pfEnableRxInterruptEP)(U8 EPIndex);
```

Parameter	Description
EPIndex	OUT-endpoint whose interrupt needs to be enabled.

Table 12.12: (*pfEnableRXInterruptEP)() parameter list

12.4.7 IN-endpoint functions

12.4.7.1 (*pfStartTx)()

Description

Starts data transfer on the given IN-endpoint.

Prototype

```
void (*pfStartTx)(U8 EPIndex);
```

Parameter	Description
EPIndex	IN-endpoint that needs to be enabled.

Table 12.13: (*pfStartTX)() parameter list

Additional information

This function is called to start sending data to the host.

Depending on the design of the USB controller, one of the following steps needs to be done:

If the USB controller sends a packet and waits for acceptance by the host, your application must:

- Enable IN-endpoint interrupt.
- Send a packet using `USB__Send(EPIndex)`.

If the USB controller waits for an IN-token, your application must:

- Enable the IN-endpoint interrupt.

12.4.7.2 (*pfSendEP)()

Description

Sends data on the given IN-endpoint.

Prototype

```
void (*pfSendEP)(U8 EPIndex, const U8 * p, unsigned NumBytes);
```

Parameter	Description
EPIndex	IN-endpoint that is used to send the data.
p	Pointer to a buffer that needs to be sent.
NumBytes	Number of bytes that needs to be sent.

Table 12.14: (*pfSendEP)() parameter list

Additional information

This function is called whenever data should be transferred to the host. Because `p` might not be aligned, it is the responsibility of the developer to care about the alignment of the USB controller buffer/FIFO.

12.4.7.3 (*pfDisableTx)()

Description

Disables IN-endpoint transfers.

Prototype

```
void (*pfDisableTx)(U8 EPIndex);
```

Parameter	Description
EPIndex	IN-endpoint that needs to be disabled.

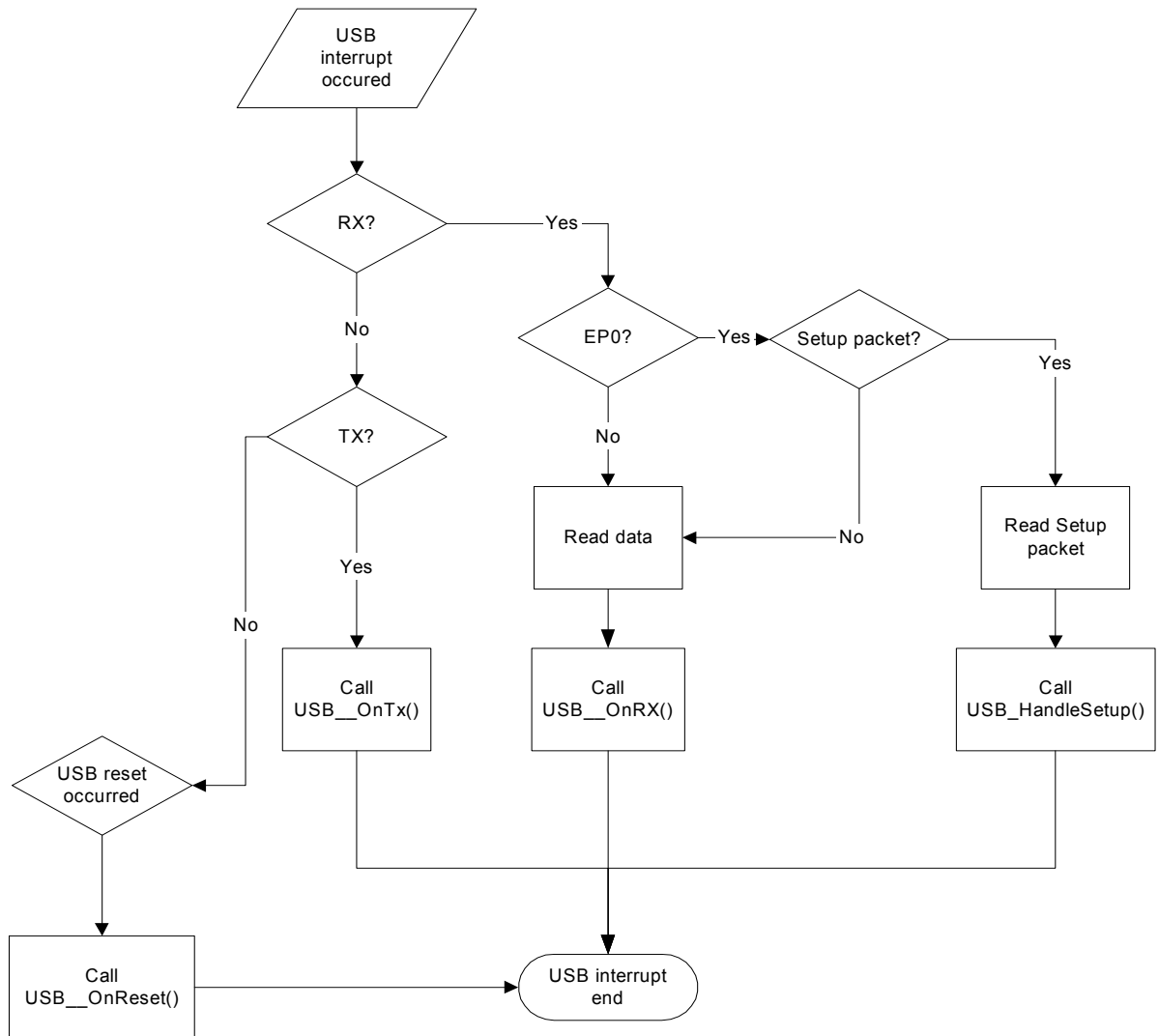
Table 12.15: (*pfDisableTx)() parameter list

Additional information

Normally, this function should disable the IN-endpoint interrupt. Some USB controllers do not work correctly after the IN interrupt is disabled, therefore this should be done by the software.

12.4.8 USB driver interrupt handling

emUSB is interrupt driven. Therefore, it is necessary to have an interrupt handler for the used USB controller. For the drivers that are available this is already done. If you are writing your own USB driver the following schematic shows which functions need to be called when an USB interrupt occurs:



Function	Description
USB_HandleSetup()	Determines request type.
USB_OnBusReset()	Flushes the input buffer and set the "_IsInReset" flag.
USB_OnTx()	Handles a Tx transfer.
USB_OnRx()	Handles a Rx transfer.
USB_OnResume()	Resumes the device.
USB_OnSuspend()	Suspends the device.

Table 12.16: emUSB interrupt handling functions

Chapter 13

Support

This chapter can help you if any problem occurs; this could be a problem with the tool chain, with the hardware, the use of the functions, or with the performance and it describes how to contact the support.

13.1 Problems with tool chain (compiler, linker)

The following shows some of the problems that can occur with the use of your tool chain. The chapter tries to show what to do in case of a problem and how to contact the support if needed.

13.1.1 Compiler crash

You ran into a tool chain (compiler) problem, not a problem with the software. If one of the tools of your tool chain crashes, you should contact your compiler support:

```
"Tool internal error, please contact support"
```

13.1.2 Compiler warnings

The code of the software has been tested with different compilers. We spend a lot of time on improving the quality of the code and we do our best to avoid compiler warnings. But the sensitivity of each compiler regarding warnings is different. So we can not avoid compiler warnings for unknown tools.

Warnings you should not see

This kind of warnings should not occur:

```
"Function has no prototype"  
"Incompatible pointer types"  
"Variable used without having been initialized"  
'Illegal redefinition of macro'
```

Warnings you may see

Warnings such as the ones below should be ignored:

```
"Integer conversion, may lose significant bits"  
'Statement not reached"  
"Meaningless statements were deleted during optimization"
```

Most compilers offers a way to suppress selected warnings.

13.1.3 Compiler errors

We assume that the used compiler is ANSI C compatible. If it is compatible there should be no problem to translate the code.

13.1.4 Linker problems

Undefined externals

If your linker shows the error message "Undefined external symbols..." check if all required files have been included to the project.

13.2 Problems with hardware/driver

If your tools are working fine but your USB-Bulk device does not work may be one of the following helps to find the problem.

Stack size to low?

Make sure there have been configured enough stack. We can not estimate exactly how much stack will be used by your configuration and with your compiler.

13.3 Contacting support

If you need to contact the support, send the following information

to support@segger.com:

- A detailed description of the problem
- The configuration file `USB_Conf.h`
- The error messages of the compiler

Chapter 14

Performance & resource usage

This chapter covers the performance and resource usage of emUSB. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

14.1 Memory footprint

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

The tests were run on a 32-bit CPU running at 48MHz. The test program was compiled for size optimization.

14.1.1 ROM

The following table shows the ROM requirement of emUSB:

Description	ROM
emUSB core	app. 5.5 KBytes
Bulk component	app. 400 Bytes
MSD component	app. 5 KBytes + sizeof(Storage-Layer)*
HID component	app. 400 Bytes
CDC component	app. 1.1 KBytes
PrinterClass component	app. 400 Bytes
USB target driver	app. 1.2 - 3 kBytes

* ROM size of emFile Storage app. 4 KBytes

Additionally 64 or 512 bytes (depends on the device speed) are necessary for each OUT-endpoint as a data buffer. This is buffer is assigned within the application.

14.1.2 RAM

The following table shows the RAM requirement of emUSB:

Description	RAM
emUSB core	app. 800 Bytes
Bulk component	app. 4 Bytes
MSD component	app. 400 Bytes
HID component	app. 30 Bytes
CDC component	app. 70 Bytes
PrinterClass component	app. 2 kBytes
USB target driver	< 1 kBytes

14.2 Performance

The tests were run on a 32-bit CPU running at 48MHz with an Atmel SAM7S device driver using the USB Bulk component.

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

Description	Speed
Bulk	
Send speed	800 KByte/sec
Receive speed	760 KByte/sec

Chapter 15

FAQ

This chapter answers some frequently asked questions.

Q: Which CPUs can I use emUSB with?

A: It can be used with any CPU (or MPU) for which a C compiler exists. Of course, it will work faster on 16/32-bit CPUs than on 8-bit CPUs.

Q: Do I need a real-time operating system (RTOS) to use the USB-MSD?

A: No, if your target application is a pure storage application. You do not need an RTOS if all you want to do is running the USB-MSD stack as the only task on the target device. If your target application is more than just a storage device and needs to perform other tasks simultaneously, you need an RTOS which handles the multi-tasking.

We recommend using our embOS Real-time OS, since all example and trial projects are based on it.

Q: Do I need extra file system code to use the USB-MSD stack?

A: No, if you access the target data only from the host.

Yes, if you want to access the target data from within the target itself.

There is no extra file system code needed if you only want to access the data on the target from the host side. The host OS already provides several file systems. You have to provide file system program code on the target only if you want to access the data from within the target application itself.

Q: Can I combine different USB components together?

A: In general this is possible, by simply calling the appropriate add function of the USB component. See more information in *Combining different USB components (Multi-Interface)* on page 249.

Index

-
- A**
- ANSI 7
- B**
- Bulk Communication
- USB_BULK_Add() 86
 - USB_BULK_CancelRead() 87
 - USB_BULK_CancelWrite() 88
 - USB_BULK_GetNumBytesInBuffer() 89
 - USB_BULK_GetNumBytesRemToRead() 90
 - USB_BULK_GetNumBytesToWrite() 91
 - USB_BULK_INIT_DATA 106
 - USB_BULK_Read() 92
 - USB_BULK_ReadOverlapped() 93
 - USB_BULK_ReadTimed() 94
 - USB_BULK_Receive() 96
 - USB_BULK_SetOnRXHook 95
 - USB_BULK_WaitForRX() 97
 - USB_BULK_WaitForTX() 98
 - USB_BULK_Write() 99
 - USB_BULK_WriteEx() 100
 - USB_BULK_WriteExTimed() 101
 - USB_BULK_WriteNULLPacket() 105
 - USB_BULK_WriteOverlapped() ... 102-103
 - USB_BULK_WriteTimed() 103
 - USB_ON_RX_FUNC 107
- Bulk Communication(Host)
- USBBULK_Close() 113
 - USBBULK_CloseEx() 114
 - USBBULK_GetConfigDescriptor() 123
 - USBBULK_GetConfigDescriptorEx() 124
 - USBBULK_GetDriverCompileDate() 121
 - USBBULK_GetDriverVersion() 122
 - USBBULK_GetMode() 125
 - USBBULK_GetModeEx() 126
 - USBBULK_GetNumAvailableDevices() . 127
 - USBBULK_GetReadMaxTransferSize() . 128
 - USBBULK_GetReadMaxTransferSizeEx() ... 129
 - USBBULK_GetSN() 130
 - USBBULK_GetWriteMaxTransferSize() . 131
 - USBBULK_GetWriteMaxTransferSizeEx() .. 132
 - USBBULK_Open() 111
 - USBBULK_OpenEx() 112
 - USBBULK_Read() 115
 - USBBULK_ReadEx() 116
 - USBBULK_SetMode() 133
 - USBBULK_SetModeEx() 134
 - USBBULK_SetTimeout() 135
 - USBBULK_SetTimeoutEx() 136
 - USBBULK_SetUSBId() 137
 - USBBULK_Write() 117
 - USBBULK_WriteEx() 118
 - USBBULK_WriteRead 119
 - USBBULK_WriteReadEx() 120
- C**
- "C" compiler 24
- "C" programming language 7
- CDC
- USB_CDC_Add() 189
 - USB_CDC_CancelRead() 190
 - USB_CDC_CancelReadEx 190
 - USB_CDC_CancelWrite() 191
 - USB_CDC_CancelWriteEx() 191
 - USB_CDC_INIT_DATA 206
 - USB_CDC_LINE_CODING 209
 - USB_CDC_ON_SET_LINE_CODING 208
 - USB_CDC_Read() 192
 - USB_CDC_ReadEx() 192
 - USB_CDC_ReadOverlapped() 193
 - USB_CDC_ReadOverlappedEx() 193
 - USB_CDC_ReadTimed() 194
 - USB_CDC_ReadTimedEx() 194
 - USB_CDC_Receive() 195
 - USB_CDC_ReceiveEx() 195
 - USB_CDC_ReceiveTimed() 196
 - USB_CDC_ReceiveTimedEx() 196
 - USB_CDC_SetLineCoding() 198-199
 - USB_CDC_SetOnBreak 197
 - USB_CDC_SetOnBreakEx 197
 - USB_CDC_UpdateSerialState() 199
 - USB_CDC_UpdateSerialStateEx() 199
 - USB_CDC_WaitForRX() 203
 - USB_CDC_WaitForRXEx() 203

- USB_CDC_WaitForTX() 204
- USB_CDC_WaitForTXEx() 204
- USB_CDC_Write() 200
- USB_CDC_WriteEx() 200
- USB_CDC_WriteOverlapped() 201
- USB_CDC_WriteSerialState() 205
- USB_CDC_WriteSerialStateEx() 205
- USB_CDC_WriteTimed() 202
- USB_CDC_WriteTimedEx() 202

- D**
- Data types 8
- Data types used

- E**
- embOS/IP
- Integrating into your system 37

- F**
- FAQ 309

- H**
- HID
- USB_HID_Add() 221
- USB_HID_INIT_DATA 224
- USB_HID_Read() 222
- USB_HID_Write() 223

- M**
- MSD
- USB_MSD_Add() 148
- USB_MSD_AddCDRom() 150
- USB_MSD_AddUnit() 149
- USB_MSD_Connect() 154, 157
- USB_MSD_Disconnect() 155
- USB_MSD_INFO 160
- USB_MSD_INIT_DATA 159
- USB_MSD_INST_DATA 161
- USB_MSD_RequestDisconnect() 156
- USB_MSD_SetPreventAllowRemovalHook()
 151
- USB_MSD_SetReadWriteHook() 152
- USB_MSD_Task() 153
- USB_MSD_WaitForDisconnection() 158

- O**
- OS
- USB_OS_DecRI() 265
- USB_OS_Delay() 264
- USB_OS_GetTickCnt() 266
- USB_OS_IncDI() 267
- USB_OS_Init() 268
- USB_OS_Panic() 269
- USB_OS_Signal() 270
- USB_OS_Wait() 271
- USB_OS_WaitTimed() 272

- S**
- Support 301–310
- Syntax, conventions used 7

- U**
- USB Core
- USB_AddDriver() 52
- USB_AddEP() 57
- USB_DoRemoteWakeup 71
- USB_EnableIAD() 68
- USB_GetState() 53
- USB_IsConfigured() 55
- USB_SetAddFuncDesc() 58
- USB_SetAllowRemoteWakeUp() 70
- USB_SetClassRequestHook() 59
- USB_SetIsSelfPowered() 61
- USB_SetMaxPower() 62
- USB_SetOnRxEP0() 63
- USB_SetOnSetupHook() 64
- USB_StallEP() 66
- USB_Start() 56
- USB_WaitForEndOfTransfer() 67
- USB_GetProductId() 43
- USB_GetProductName() 45
- USB_GetSerialNumber() 46
- USB_GetVendorId() 42
- USB_GetVendorName() 44
- USB_HW_DRIVER
- (*pfAllocEP)() 292
- (*pfAttach)() 290
- (*pfDisableRxInterruptEP)() 296
- (*pfDisableTx)() 297
- (*pfEnable)() 290
- (*pfEnableRxInterruptEP)() 296
- (*pfGetMaxPacketSize)() 292
- (*pfInit)() 289
- (*pfSendEP)() 297
- (*pfSetAddress)() 290–291
- (*pfSetClrStallEP)() 293–294
- (*pfStallEP0)() 295
- (*pfUpdateEP)() 293
- USB_HW_StartTx() 298
- USB_Init() 54
- USB_MSD_INST_DATA_DRIVER 164
- USB_MSD_STORAGE_API 165
- (*pfDeInit)() 174
- (*pfGetInfo)() 168
- (*pfGetReadBuffer)() 169
- (*pfGetWriteBuffer)() 171
- (*pfInit)() 167
- (*pfMediumIsPresent)() 173
- (*pfRead)() 170
- (*pfWrite)() 172
- USB_SetAllowRemoteWakeUp 70
- USBHID_Close() 227
- USBHID_Exit() 230
- USBHID_GetInputReportSize() 233
- USBHID_GetNumAvailableDevices() 231
- USBHID_GetOutputReportSize() 234
- USBHID_GetProductId() 235
- USBHID_GetProductName() 232
- USBHID_GetVendorId() 236
- USBHID_Open() 228
- USBHID_RefreshList() 237
- USBHID_SetVendorPage() 238